Vrije Universiteit Brussel

FACULTEIT VAN DE WETENSCHAPPEN
Vakgroep Informatica
Programming Technology Lab

Ambient-Oriented Programming

Ph.D. Dissertation

Jessie Dedecker

Promotors: Prof. Dr. Theo D'Hondt and Dr. Wolfgang De Meuter

23 May 2006

Abstract

As a result of the computing technology that becomes ever smaller and cheaper it is now possible to integrate it into everyday material objects. This advanced integration of technology allows the underlying computer to disappear into the fabric of life so that by manipulating material objects we are transparently interacting with the underlying integrated technology. The invention of wireless communication technology enables these disappearing integrated computers to cooperate with one another so that they can derive context about its environment. The advantage is that users can be supported more naturally and transparently to achieve their goals. This vision is often referred to as "Ambient Intelligence" (AmI).

The research presented in this dissertation deals with the problem of software development for these invisible computers from the perspective of distributed systems. Developing software for such systems is difficult because of inescapable characteristics exhibited by the hardware. For example, as a consequence of the use of wireless communication media connections can break at any point in time due to interference in the environment and the mobility of material objects. To address these hardware phenomena at the software level we propose a new programming paradigm called "Ambient-Oriented Programming" (AmOP). This programming paradigm is derived from the most important hardware phenomena.

The next step in this dissertation is to gain insight in the structure of AmOP applications. Although the definition of a paradigm is a first step towards this goal, it is insufficient to derive the structure of AmOP applications. To gain insight in the structure of AmOP applications it was necessary to experiment with new language features. The definition and experimentation with new language features is necessary for three reasons: 1) it supports the developer to capture the consequences of the hardware phenomena in the code. 2) without proper language features the integration of the AmOP paradigm with the object paradigm leads to complex program structures. 3) at this point there is not enough experience in building applications that enable AmI scenarios.

To support experiments with language features we build an AmOP programming language. The first step towards such a programming language is the choice of a concurrency and distribution model, which we defined as a formal extension of the actor model. This formal model serves as a base for the concurrency and distribution model of an AmOP kernel language, called AmbientTalk. AmbientTalk is a little reflectively extensible language that supports experimentation with new language features. New language features are defined in AmbientTalk itself out of semantic building blocks, which are shaped by the AmOP paradigm. These semantic building blocks are used to extend

AmbientTalk with existing and new language features. These language features support the developer in addressing the inescapable consequences of the hardware phenomena.

Acknowledgements

This dissertation would not have been what it is today without the tremendous support that I have received from my colleagues, friends and family.

I would like to thank Theo D'Hondt not only for inspiring me to do research but also for providing me with the means to do it. Besides having sparked my interests for research Theo also introduced to the EMOOSE master program, which has both enriched me from an intellectual and social perspective.

A BIG thank you also goes to Wolfgang De Meuter for being there during each step towards this dissertation and for providing me with all those useful comments, tips and "pep talk" at the right moments.

I thank the members of my thesis committee, Prof. Cristina Videira Lopes, Prof. Wouter Joosen, Prof. Viviane Jonckers and Prof. Bernard Manderick, for comments on the first version of the text.

Two other people I am greatly indebted to are Tom Van Cutsem and Stijn Mostinckx. Tom helped me with some of the implementations of the experiments, meticulously checked all technical details and provided me with useful comments on how to improve the readability. Stijn proofread some of the technical chapters and generously took over my teaching responsibilities while I was writing. Wolfgang, Tom and Stijn not only helped me enormously while I was writing but were also my partners in crime during the last couple of years. I look forward to continue working with them in the future.

I also thank Werner Van Belle with whom I developed the formal actor extensions in this dissertation. Werner also helped me with finding a focus in the first year, which was very important in order to finish this dissertation within the time constraints of my funding.

Thanks also to all the people who helped in improving the quality and clarity of my writing by proofreading and commenting on preliminary versions of this dissertation. Wolfgang De Meuter, Tom Van Cutsem, Pascal Costanza and Stijn Mostinckx have helped a lot me to improve the quality of the text.

Peter Ebaert and Elisa Gonzalez Boix also deserve special mentioning. Peter took care of my responsibilities concerning the EMOOSE program while I was writing. Elisa started developing a concurrency extension for Pico during her training at PROG. This extension was employed in the prototype used to conduct the experiments in this dissertation.

I also thank the other members of our lab for providing me with useful comments at my research meetings and for enduring me at all those moments I was (unreasonably?) stressed: Andy Kellens, Brecht Desmet, Coen De Roover, Dirk Deridder, Dirk van Deun, Ellen Van Paesschen, Isabel Michiels, Johan Brichau, Johan Fabry, Jorge Vallejos Vargas, Kris Gybels, Linda Dasseville, Sofie Goderis, Thomas Cleenewerck.

Thanks to my friends Tim Dobbelaere, Bram Bruneel, Henk Brouckxon, Lies Van Doren, Sebastián González, Agustina María Cibrán, Boris Mejias, Werner Van Belle, Antoon Goderis, Michael Vernaillen, Bert Schiettecatte and Frank van der Kleij for supporting me during the years towards this dissertation.

I want to thank the secretaries, Lydie Seghers, Brigitte Beyens and Simonne De Schrijver for helping me out with all the administrative issues. Also thanks to Philippe Debroey, who made a professional graphical design of the AmbientTalk logo.

Thanks to my girlfriend Ellen Degreef for her enormous support and for enduring me for the last couple of months, when I was mainly concerned with my dissertation. Another big thank you goes to my parents, who have always supported me and my interest in technology long before I went to university.

Finally, I wish to thank everyone for having had the opportunity to write this dissertation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Although computing technology is currently omnipresent in the modern world, it has not become invisible at all. Today, people to a large extent have to interact with their computer in order to get certain tasks done. Ubiquitous computing is a vision, postulated by Mark Weiser [Wei91]: computing technology will become invisible as it is integrated into the fabric of everyday life. This vision has been termed *Ambient Intelligence* by the European Council's IST Advisory Group [IST03]. Weiser explains the invisibility of a computer by making the analogy to text. A world without text is unthinkable in our modern society. Text allows ideas, which were previously perhaps mere thoughts in the brain of a single person, to be made persistent accurately such that knowledge and culture can survive time. Today, text has become ubiquitous: from books to the internet and from traffic signs to button labels on a remote control. For most people the act of reading text has become oblivious, because in modern societies it is learnt from childhood on. As a result people are no longer consciously aware of text such that, in a sense, text has become invisible to people. According to Weiser computing technology is about to make the same leap. The first signs of this transition are already present in many current-day products. Television sets and video recorders have become fully digital even though the user can still interact with them in much the same way as when they were analog. Contemporary cars have an enormous amount of electronics embedded in them and many of these are invisible to the driver. These electronics can change the parameters of the engine to adapt itself to current driving conditions or even send a self-performed check to the car dealer. The dealer can analyze these results and invite the driver for maintenance of his car should that be necessary. Although these examples are somewhat modest they show that the user is no longer confronted with the technology embedded in the products. Nevertheless, when considering the last example it shows that *interaction* of devices can further make certain processes transparent. Indeed, before, the driver had to be consciously aware of the maintenance schedule of his car whereas because of the interaction of his car with his dealer this is now automatically taken care of. This is a first example how oblivious interactions between material objects can lead to further making technology invisible.

The vision postulated by Mark Weiser has nowadays become technically feasible because of continuous and recent developments in technology.

Miniaturization of Hardware Over the past decades we have seen that hardware became ever smaller. This miniaturization of hardware is necessary in order to embed computing technology into everyday objects. Also on the level of power consumption a lot of progress has been made. Perhaps this trend becomes most concrete when we compare the autonomy of contemporary cellular phones to those of a decade ago. Phones nowadays have a battery autonomy of up to almost two weeks compared to only a few hours initially, whereas their functionality has increased. New battery technologies based on hydrogen fuel cells with an even larger capacity are making their way into the market.

Wireless Communication The advent of wireless communication technology (such as WiFi, Bluetooth, 802.15.x and others) signifies a big step towards the realization of this vision, because it enables material objects to interact autonomously with one another. Furthermore, the fact that these objects need not be connected via a wire also adds to the invisibility of technology for its users because their mobility is not hampered. Wireless communication also enables objects to detect other objects that are located in their immediate proximity. As noted by Weiser this single ability allows software to distill facts about its environment that it was previously unable to do. For example, suppose a meeting is ongoing in a meeting room. The cellular phones of the participants of this meeting could detect that they are in the same room, because the meeting room, which is equipped with wireless communication technology, can spontaneously interact with the cellular phones. Also, the video projector which is turned on to accommodate the audio-visual aspects of the meeting indicates that a meeting is ongoing. Due to the fact that the cellular phone can spontaneously interact with the meeting room and the video projector allows it to adapt its configuration such that incoming calls do not disturb the meeting. Note that such intelligent adaptations do not necessarily rely on artificial intelligence techniques. They simply result from the *spontaneous interaction* that is enabled by wireless communication technology.

1.1 Research Context and Motivation

Before we dive into the research context of this dissertation we make the vision of ambient intelligence and ubiquitous computing more concrete by giving a concrete scenario in which the vision has been realized. Next we explain the problems associated with the realization of this vision and set the research goals for this dissertation.

1.1.1 A Futuristic Scenario

It is the year 2020 on a sunny monday at 05:30 in the morning. Theo is still in a deep sleep. His first appointment for the day is at 10:30 in the morning but his family has the habit to start the day with a good breakfast sitting all together. For this reason the digital alarm clock consults the agenda of all the family members and calculates that Theo's daughter Maja has the earliest appointment. She needs to be at school at eight in the morning and the route planner indicates based on the traffic predictions over the past year that she needs half an hour to reach school in time. Based on this information the digital

alarm clock decides that it is time to wake up all family members such that they can have breakfast together. The digital alarm clock opens the curtains of all the bedroom windows such that the sunlight gently starts to wake up all the family members and the coffee machine automatically starts making coffee. Half an hour later, at six in the morning, the digital alarm clock notices, through interaction with the mattress, that Theo is still in bed in spite of the opened curtains. As a result it starts to play Theo's favorite songs to further wake him up. Five minutes later Theo arrives in the kitchen and joins his family at the table. Theo's daughter, Maja, takes the last box of corn flakes from the pantry and eats the last flakes that are in the box. They are talking about the things they did together during the past weekend and their plans for the day. Theo reads the newspaper with his cup of coffee and scans through the television program section. He circles the programs that interest him with his ball pen. The ball pen communicates the selections to his digital media manager that automatically schedules the programs for recording.

After breakfast Theo cleans up the kitchen table while everyone else goes on to prepare themselves for the day. Theo throws the empty box of corn flakes in the garbage can. The garbage can notifies the other objects in its environment of this event. The pantry, which is responsible for stocking the corn flakes, notices that it no longer holds any stock of corn flakes and decides to place it in the family's electronic shopping cart.

Later on this morning Theo drives to his first appointment of the day and drives past a super market. His electronic shopping cart notices this and communicates with the super market. Coincidentally there is a sale for corn flakes: three bags plus one for free. The shopping cart consults the agenda of Theo and checks with his car's GPS system and discovers that Theo is scheduled to arrive half an our early at his first appointment, so it decides to notify Theo of this bargain. However, Theo has made other plans for the day and presses a button on his PDA to have the corn flakes delivered at home and he continues his way.

1.1.2 What is the Problem?

The scenario above illustrates what a world where ubiquitous computing has been realized could look like. Many of the hardware problems associated with the realization of this vision have already been solved. In fact, the hardware components to realize most of the ubiquitous computing scenarios are already available on the market today. Nevertheless, very few ready-made products based on this vision are available to consumers.

> It is our conjecture that such scenarios have not been realized because although the hardware technology is available the software technology is actually incapable to manage the complexity of the dynamics of ubiquitous computing.

This lack of support today is witnessed by the numerous recent and future workshops on this topic in the year 2005 and 2006 only: "Smart Object Systems" [KLM+05], "Object Technology for Ambient Intelligence" [MCCH05, MCMT06], "Building Software for Pervasive Computing." [LSC+05], "Software Engineering of Pervasive Services" [CSM+06] and others. These workshops also

underline that the existing object-oriented paradigm, which is currently the most successful paradigm for building large software systems, does not provide sufficient support to construct these systems.

Much of the emerging behavior in ubiquitous computing scenarios results from the cooperation between devices. These devices can cooperate because they are surrounded by what is sometimes referred to as a *mobile network*. A mobile network emerges from a set of devices that communicate over wireless communication media. This type of network has several properties that distinguish it from other types of networks. The most important ones are that connections are volatile (because the communication range of wireless technology is limited) and that the network is open (because devices can appear and disappear unexpectedly). This puts extra burden on software developers. Although low-level system software and networking libraries providing uniform interfaces to the wireless technologies (such as JXTA [Gon02] and M2MI [KB02]) have matured, developing application software for mobile networks still remains difficult. One of the main reasons for this is that traditional programming languages capture failing remote communication using a classic exception handling mechanism. This results in application code polluted with exception handling code because failures are the rule rather than the exception in mobile networks. The above conjecture can thus be reformulated by stating that contemporary programming languages lack the abstractions to deal with mobile hardware characteristics.

1.1.3 Research Goals

Observations like this justify the need for a new *Ambient-Oriented Programming* paradigm (AmOP for short) that supports programming languages that explicitly incorporate potential network failures in the very heart of their basic computational steps. Although a paradigm is a first necessary step towards the supporting the design and development of AmOP applications it does not necessarily give insight into how such applications are built and how different concerns should be expressed. Similarly, design patterns for the object paradigm were only captured decades after the paradigm was invented. What is more, the range of possible applications is broad and it is not yet clear what types of applications will be built based on the new hardware. However, it is only when good software technology becomes available that advanced applications will be developed. Hence, we are faced with a chicken and egg problem. As a consequence the goal of our research is threefold:

- First, we want to come up with AmOP language features that give programmers expressive abstractions that allow them to deal with the characteristics of mobile networks.
- Second, we want to gain insight in the structure of AmOP applications.
- Third, we want to distill the fundamental semantic building blocks that are at the scientific heart of AmOP language features in the same way that current continuations are at the heart of control flow instructions and environments are the essence of scoping mechanisms.

These three research goals allow us to bootstrap this cycle and address the software technology problem we are faced with.

1.2 The Thesis

This dissertation demonstrates that languages supporting the "Ambient-Oriented Programming" paradigm can better support the development of AmOP applications by providing language features to address specific issues arising from the hardware used to construct AmI scenarios. These language features reconcile object-oriented programming methodology with the ambient-oriented programming paradigm.

The thesis is validated with the following results:

1. The definition of a collection of language features constructed based on the defining characteristics of the paradigm. Some of these language features transpose well-acknowledged concepts to deal with distributed issues into an AmI-context and support an object-oriented programming style.

2. A study of the use of language features in a concrete application. This application is compared both qualitative (in terms of how issues are dealt with) and quantitative (in terms of lines of code) to a similar application written in a language not based on the paradigm.

1.3 Problem Statements

Above, we already stressed that much of the emerging behavior of ubiquitous computing systems result from cooperation of devices that reside in each others ambient. However, before devices can cooperate they need to discover one another in their ambient. Once they have discovered one another a connection between devices can be established and devices can communicate. Communication between devices enables cooperation, such that each of these devices can interact and respond to events that occur in the ambient. However, interactions and events in the ambient are concurrent because they result from events in the "real world".

This exposition led us to define four problems that will have to be addressed in order to work towards our research goals. These problems are further elaborated below:

- **Problem #1: Object Model for AmOP**
  Since our approach will be based on the paradigm of object-oriented programming we will have to come up with an object system that supports the development of AmOP applications. Current mainstream programming languages are all based on class-based object systems. In this dissertation we will argue that classes are both conceptually and technically a source of problems. Furthermore, we will argue that classless languages do not suffer from these problems.

- **Problem #2: Concurrency Control for AmOP**
  Another major issue is concurrency control for AmOP applications. AmOP applications will collaborate and cooperate with other AmOP applications that are in the ambient of the device. These devices are often embedded into the "fabric of everyday life" and as a consequence they will interact with the world. The world is inherently concurrent and in order to adequately support these interactions AmOP applications will need to deal

with the concurrency that results from it. Traditionally concurrency control is most often expressed such that inconsistent states are prevented. Such an approach is often classified as pessimistic concurrency control. The antipode, optimistic concurrency control, is to allow inconsistencies to occur and afterwards to repair the inconsistent state should that be necessary. In [Sat96] it is determined that mobile computing applications need support for optimistic concurrency control. However, current approaches that support optimistic concurrency control are rarely object-based.

- **Problem #3: Communication Mechanisms for AmOP**
  We already stated that volatile connections are a fundamental characteristic that result from wireless communication. An obvious problem is that we will have to devise a communication mechanism that is able to cope with the volatility that results from the use of wireless communication media. A problem related to this is what is called the "graceful degradation" of a system [FRBAM05] . What is meant by this is that resources that are suddenly no longer available should not hamper the full system. Instead the system should be able to continue to work in spite of the resource no longer available, albeit perhaps in a degraded form. An issue directly related to the graceful degradation of a system is the ability to deal with disconnectedness. Groups of devices can become isolated from other devices resulting in network partitions. These network partitions can result in conflicts after these groups of devices regain their connectivity. Hence, support is needed to resolve such conflicts.

- **Problem #4: Ambient Resource Management for AmOP**
  Much of the potential of ubiquitous computing is realized by the potential to cooperate with devices in the ambient. However, the ambient is a volatile concept because computing technology is integrated in all kinds of objects such as clothes, furniture and cars. Hence, many of these are subject to frequent movement and a result the devices in the ambient change continuously. As a result AmOP applications need to deal with this constant change of available resources. A problem related to this is that applications should support "self-configurability" [FRBAM05]. Hence, AmOP applications should be able to autonomously react to changes in the ambient and be able to reconfigure themselves based on the available resources.

1.4 Research Approach

During the course of this dissertation we had to make a number of choices with regard to the development of a number of experiments. This section motivates these choices and explains how we have approached the problems described above.

1.4.1 An Experimental Approach

As very limited experience exists in writing AmOP applications, it is hard to come up with AmOP language features based on software engineering requirements. Therefore, our research starts from the hardware phenomena that fundamentally distinguish mobile from stationary networks. These phenomena are

based on the properties of the hardware that is used to run AmOP applications. They form the basis from which we distill a number of fundamental characteristics that define the AmOP paradigm, thereby addressing the third goal of our research.

Based on the abstract characteristics of this new paradigm we will design and develop a concrete prototype. This prototype will allow us to develop concrete AmOP applications and gain insight into their structure. Nevertheless, one could argue that the structure of AmOP applications will depend on the applications themselves. Although this is certainly true to a certain extent we will find that much of the structure of the applications also results from the AmOP paradigm itself. What is more, the AmOP paradigm is based on the properties exhibited by the hardware components, on which the majority of the AmOP applications are built, such that part of the structure will return in many AmOP applications. Hence, by the development and experimentation with this first prototype of the paradigm we address the second goal of our research.

In order to address our first research goal, namely the development of suitable language features for the AmOP paradigm, we based our research on two sources of information. A first source was the experience we have built thus far based on the implementation of some AmOP applications. A second source of information was the structure and goals found in the current state of the art. In the state of the art we have found many interesting abstractions to resolve specific problems. Hence, we have based the design and development of the AmOP programming abstractions based on these two sources of experience.

What is more, the design and development of object-oriented versions of these abstractions found in the current state of the art also served as a validation of the expressiveness of the AmOP paradigm and its first concrete prototype. Furthermore, some of the abstractions we designed and developed were used in a concrete experiment we conducted.

1.4.2 Languages vs. Middleware

In the design and development of a system that is to support distributed programming a first choice one has to make is to make the abstractions available as a programming language or as middleware. Today most researchers have conducted their experiments based on middleware. Middleware is often said to have the advantage that it is easily usable in a "real world" setting such that the results from research can more easily be adopted. Although this is true to a certain extent there are a number of limitations associated with a middleware approach:

- Bal et al. [BST89] described a number of reasons why the language approach is better than a library approach. Languages are often better at expressing concerns that relate to concurrency and distribution. They note that expressing complex synchronization constraints in many libraries requires multiple lines of code compared to a single line of code in a distributed language. Readability is another advantage that is somewhat related to the improved expressiveness. Finally, they note that the fact that languages offer higher-level abstractions is probably the most important reason. However, since the time Bal et al. [BST89] surveyed the state of the art some issues have somewhat been alleviated through the use of

reflection and advanced preprocessor techniques. Nevertheless, these techniques are currently not well supported by mainstream languages and do not resolve all of the issues. For example, interactions can occur with existing language features.

- More recently, Varela and Agha [VA01] came to a similar conclusion. They noted that a programming language had the following advantages:
  - **Semantic Constraints**: distributed languages are better at enforcing properties of the underlying model for concurrency and distribution, because they have total control over the execution of a program. Hence, distributed languages are often better because they can preserve and promote the paradigm supported by the language.
  - **API Evolution**: in a distributed language important concepts of the supported paradigm are aligned with language concepts such that they eliminate the reliance on an API for using these paradigm concepts in the language. This alignment of concepts often ensure that programmers cannot incorrectly use the concurrency and distribution APIs. What is more, local changes made to the underlying model of the programming language often need not affect the existing code of applications already written in the language.
  - **Programmability** This point refers to the increased expressiveness and readability of languages. They also note that in a library it is often necessary to follow a specific protocol which is implicitly defined by the API in order to perform primitive operations. An example of such a protocol is that a stream of data must be opened before data can be read or written and must be closed after all data has been read or written. Such conventions are often not enforced and as a result there is a need for permanent semantic translation, which is unnatural for programmers.

The advantages described above are all from the developer's point of view. What is more, it is our conjecture that the language based approach offers more scientific potential from a research perspective. The reason is that the language-based approach is not constrained with respect to the expressiveness of the underlying language. Although current mainstream languages have certainly benefited from research over the past decades, such as the introduction of automatic memory management, they still impose a number of restrictions. For example, Java does not support closures and it is only recently that C# has introduced support for them. These and other constraints fence off the possibilities and can limit the design and introduction of new abstractions.

1.4.3 Implicit vs. Explicit Distribution

In the past many approaches towards distributed programming models were based on the fact that distribution should be transparent for the application at hand. Jim Waldo et al. wrote a paper [WWWK96] on the reasons why such a goal does not scale to the problems that are encountered in distributed systems. Vogels et al. came to a similar conclusion [VRB99] and adjusted their research goals accordingly. The main reasons for the need for explicit distribution is

that distributed systems have a number of properties that distinguishes them from local computation. The important ones are that distributed systems are subject to a communication latency that varies based on the type of network used and on the load of the network. Second, distributed systems have a different model of memory access based on low-level message sending rather than local memory access. Finally, distributed systems can be concurrent and subject to partial failure. These properties distinguish a local computational model from a distributed computational model. Merging the models by making local computation follow the model of distributed computing makes local programs unnecessarily complex, whereas making the model of distributed computing follow the local computation model requires ignoring the distributed properties such failing nodes.

In the context of mobile networks, where failing connections are the rule rather than the exception, one will soon run into the issues that are described in these papers. For this reason we have chosen to make the concerns of distribution explicit with respect to the application. As noted by Jim Waldo et al. [WWWK96] a better approach is to accept that there are irreconcilable differences between local and distributed computing, and to be conscious of those differences at all stages of the design and implementation of distributed applications. This choice was taken into consideration throughout the whole research process: from the definition of the paradigm to the conception of the distributed language. For example, one such place is the syntax of the language that makes potentially distributed calls explicit. The fact that distribution is made explicit does not mean that one has to continuously deal with *all* intricacies of distribution. Hence, the art is to find abstractions that allow one to manage the complexity of distribution concerns without losing the ability to deal with the consequences of their presence.

1.4.4 Language Design Choices

Several methodologies exist for designing a language. Richard Gabriel distinguished between the MIT/Stanford approach and the New Jersey approach in his paper [Gab91] often referred to as "Worse is Better". He characterizes the MIT/Stanford approach as follows:

> The essence of this style can be captured by the phrase "the right thing." To such a designer it is important to get all of the following characteristics right:
>
> - Simplicity: the design must be simple, both in implementation and interface. It is more important for the interface to be simple than the implementation.
> - Correctness: the design must be correct in all observable aspects. Incorrectness is simply not allowed.
> - Consistency: the design must not be inconsistent. A design is allowed to be slightly less simple and less complete to avoid inconsistency. Consistency is as important as correctness.
> - Completeness: the design must cover as many important situations as is practical. All reasonably expected cases must be

covered. Simplicity is not allowed to overly reduce completeness.

Richard Gabriel characterized the New Jersey approach as follows:

> The worse-is-better philosophy is only slightly different:
>
> - Simplicity: the design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design.
> - Correctness: the design must be correct in all observable aspects. It is slightly better to be simple than correct.
> - Consistency: the design must not be overly inconsistent. Consistency can be sacrificed for simplicity in some cases, but it is better to drop those parts of the design that deal with less common circumstances than to introduce either implementational complexity or inconsistency.
> - Completeness: the design must cover as many important situations as is practical. All reasonably expected cases should be covered. Completeness can be sacrificed in favor of any other quality. In fact, completeness must sacrificed whenever implementation simplicity is jeopardized. Consistency can be sacrificed to achieve completeness if simplicity is retained; especially worthless is consistency of interface.

He further argues in his paper that the MIT/Stanford approach is the right design philosophy, but that the New Jersey philosophy has certain properties that make it more popular. In essence, it boils down to the fact that a language designed according to the MIT/Stanford approach takes too long to conceive. This is in contrast to the New Jersey approach that has a shorter development cycle. The past has proven that developers and users do not want to wait for "the right thing". Nevertheless, since this paper Richard Gabriel has written several follow-up papers [Gab00] where he reconsiders his point of view multiple times. In the last paper of the series he gave up and wrote "Decide for yourselves."

In the conception of the programming language we have tried to adhere to the MIT/Stanford design approach, which is the reason why we have decided to build a language from the ground up instead of extending an existing language. Unfortunately the result cannot yet fully be classified as a language with the properties of the MIT/Stanford approach. However, in retrospect it is not such a surprise because the language was primarily conceived as a language laboratory to facilitate experimentation with language constructs for AmOP applications. It is only after we have identified the correct abstractions and gained sufficient experience in building AmOP applications that we will be able to build that kind of language.

Our approach started from the design philosophy of little languages [Ben86]. We have designed and implemented a small kernel language, called AmbientTalk, that facilitates the AmOP paradigm. AmbientTalk formed the basis for further experimentation with language features. At that point we crossed over to the philosophy to let AmbientTalk grow [GLS98]. This was facilitated

by opening up the implementation of the kernel language through a reflective interface. Hence, AmbientTalk was conceived as a reflectively extensible language kernel with semantic building blocks that turned it into a language laboratory so as to allow us to investigate the language features that populate the AmOP paradigm.

1.5 Contributions

In this section, we summarize the major contributions of this thesis:

1.5.1 AmOP Paradigm

One of the major contributions is to promote the AmOP paradigm for programming mobile distributed systems. The AmOP paradigm defines a set of criteria for concurrent distributed object-oriented languages such that they can better support the hardware phenomena exhibited by mobile distributed systems.

The AmOP paradigm is defined by a set of four defining characteristics for concurrent distributed languages. Each of these individual AmOP characteristics are not entirely new. For example, one of them, non-blocking communication primitives, is exhibited by some event-driven systems. However, to the best of our knowledge no programming language or middleware exists that combines the four characteristics in a single coherent platform. The combination is important because, as discussed in chapter 3, each of these individual characteristics are necessary to reduce the complexity of the software and address the issues induced by the hardware phenomena used to construct AmI scenarios.

1.5.2 The Ambient Actor Model

Another contribution made by this dissertation is the formal extension of the actor model. The actor model is one of the first true concurrency models created for the object paradigm. It is the basis for today's state of the art concurrency and distribution models. The ambient actor model extends the actor model such that it supports the AmOP paradigm with the introduction of first-class mailboxes. These first-class mailboxes are part of semantic building blocks used to construct different language features to better support AmOP applications.

1.5.3 Language Experimentation Laboratory

In this dissertation we have defined and specified a reflectively extensible language kernel based on the ambient actor model that enables experiments with AmOP language features. We argue that reflection is a suitable implementation mechanism for experimental AmOP language features. Reflection allows one to define the language constructs within the language that is being extended. This has the advantage that the technical overhead to implement and experiment with language constructs is minimized. What is more, the native implementation of the language that is implemented can be reduced to a small kernel language with a minimum of concepts. This not only reduces the complexity of the native implementation but also results in a language with uniform semantics. Another advantage of this approach is that the language constructs can more easily be adapted to fit the needs of the application.

1.5.4 AmOP Language Constructs

Although some of the language constructs proposed in this dissertation are heavily inspired by language constructs found in other languages, some of them are entirely new. Ambient references promote the use of object references to address ambient resources and thereby unify communication with service discovery. The language construct `due` allows one customize the delivery of messages based on application-specific needs.

In addition to the basic language constructs we also propose language constructs to address advanced issues when building mobile distributed systems. The group communication abstractions allow for the coordination of multiple concurrent tasks in a setting of mobile distributed systems. Perhaps the most innovative language construct is the use of reversible computation to enable weak replication at the object level.

These language constructs not only demonstrate the potential of the AmOP paradigm, they also illustrate that the AmOP paradigm can be reconciled with an object-oriented style for programming and structuring AmOP applications.

1.6 Roadmap

In section 1.4.1 we have described how we will work towards the research goals we have set in section 1.1.3. The overall outline of this dissertation matches that description:

- Chapter 2 starts by considering two types of distributed systems that can be built from the hardware components used to construct ubiquitous computing scenarios. Based on the properties of these hardware components we distill the fundamental hardware phenomena and these are used to evaluate distributed programming languages and middleware specifically designed for mobile distributed systems.

- Chapter 3 considers these hardware phenomena again and uses them as the yardstick for the design criteria to craft a new AmOP paradigm that supports the development of AmOP applications. Furthermore, we consider the problems we observed in the state of the art again with respect to their support for the AmOP paradigm.

- Chapter 4 considers the actor model in the context of the AmOP paradigm. We will come to conclude that the actor model does not support the AmOP paradigm but nevertheless satisfies a number of properties that are also found in the AmOP paradigm. Based on these observations an extension of the operational semantics of the actor model is proposed to align the actor model with the AmOP paradigm. The extension is then used to express a first application in the paradigm. Finally, we will come to conclude that the direct applicability of the extension is limited because the applications are expressed in the lowest-level building blocks of the paradigm.

- In chapter 5 we consider the operational semantics as a low-level kind of instruction set that complies with the AmOP paradigm but which lacks sufficient support to express high-level AmOP programming abstractions

and applications. Based on this observation we design and informally describe a full-fledged high-level ambient-oriented language, called AmbientTalk, based on this low-level instruction set. Although this language is already higher-level than the operational model there are still some limitations with respect to the extensibility and support to develop high-level programming abstractions.

- In chapter 6 we define AmbientTalk's exact semantics based on a metacircular definition. This metacircular definition is then used as a basis to open up parts of the native AmbientTalk implementation. More particularly we use it to specify AmbientTalk's reflective hooks on its model of concurrency and distribution. This renders AmbientTalk as a reflectively extensible language kernel. We also further investigate a number of engineering principles to define metaprograms in AmbientTalk.

- At this point in the dissertation we have arrived at a language that is expressive enough to capture and modularize programming abstractions for AmOP applications. In chapter 7 we exploit this expressiveness and start to capture additional programming abstractions as language features. As such we create a set of language abstractions that allow us to structure AmOP applications in the AmOP paradigm. Finally, we apply some of these abstractions in the context of an application and compare the results to a similar application written in Java.

- In chapter 8 we further demonstrate the expressive power of AmbientTalk and discuss the implementation of a number of language constructs ranging from group communication to support for reversible computations in the context of mobile networks.

- Chapter 9 summarizes and presents our conclusions found in this dissertation. It also discusses future work generated by this dissertation.

Chapter 2

Software Platforms for Mobile Distributed Systems

2.1 Introduction

Much of the emerging behavior in ubiquitous computing scenarios results from the cooperation between devices. These devices can cooperate because they are surrounded by what is sometimes referred to as a *mobile network*. A mobile network emerges from a set of devices that communicate over wireless communication media. The systems that result from such a hardware constellation are called *mobile* distributed systems. A mobile distributed system explicitly supports *mobile computing*. Mobile computing concerns the computation that is carried out in mobile devices. Mobile computing should not be confused with mobile computation, which concerns the mobility of code between devices. In this dissertation we focus on mobile computing.

In the next section we consider the characteristics of two types of such mobile distributed systems. These two types of mobile distributed systems are composed of the same type of hardware components. From the properties of these hardware components we identify important phenomena in section 2.3. These phenomena are fundamental because of their consequences to the concepts of concurrency and distribution. These concepts and their impact on aspects that need to be considered when developing concurrent and distributed software in the context of mobile distributed systems are discussed in section 2.4. In section 2.5 we review how the object-oriented paradigm can be used to model and structure concurrent and distributed applications based on these concepts. This paradigm has proven its merit with respect to dealing with distribution (and its induced concurrency) because it successfully aligns encapsulated objects with concurrently running distributed software entities [BGL98].

The remainder of this chapter will be spent at the evaluation of distributed programming platforms in the context of mobile distributed systems. One of the evaluation criteria is how they integrate with the object-oriented paradigm. For this reason we have made an explicit distinction between object-oriented distributed languages and middleware. Object-oriented distributed languages, discussed in section 2.6, are designed to unify concurrency and distribution concepts with the object-oriented paradigm. On the other hand, middleware

platforms, which are discussed in section 2.7, do not necessarily integrate well with the object-oriented paradigm. In these sections we will come to conclude that the distributed language approaches integrate well with the object-oriented paradigm, but do not support all the hardware phenomena. On the other hand, middleware approaches do not integrate well with the object paradigm, but provide better support to deal with the hardware phenomena.

2.2 Types of Mobile Distributed Systems

In this section we examine the commonalities and the differences of *fixed* and *mobile* distributed systems by means of a conceptual framework developed by Mascolo et al. [MCE02].

A general definition that encompasses both types of distributed systems is given by Coulouris et al. [CDK05]:

Definition 1 (Distributed System) *A distributed system consists of hardware and software components located at networked computers that communicate and coordinate their actions only by message passing*[1].

From this definition we can zoom in on three facets of distributed systems:

- **Type of Device**: In the definition above the term "networked computer" can refer to a *fixed* device or a *mobile* device. Fixed devices range from desktop computers and server racks to electronics embedded in stationary objects such as a washing machine. On the other hand, mobile devices can vary between laptops, PDAs, mobile phones and other electronics embedded into mobile items, such as a wrist watch.

- **Type of Network Connection**: The word "communication" refers to the network infrastructure and this is the basis for another difference between fixed and mobile of distributed systems. On the one hand, in fixed distributed systems computers are often connected via permanent links. These links are often high-bandwidth and supported by redundant infrastructure such that connections are relatively stable. Hence, disconnections are either caused by scheduled maintenance or unforeseen failures. On the other hand, mobile distributed systems are usually connected via a wireless communication link over wireless technologies such as Bluetooth, Wireless Fidelity and GPRS. These wireless technologies are prone to disconnections due to the limited communication range of these technologies. When users move about with their mobile devices they leave and enter the communication range of other devices in the environment, but even when two wirelessly communicating devices are stationary the link can be broken due to a radio occlusion caused by the environment, such as a car that passes in between the two communicating devices. The communication range is often further reduced by the limitations of the power source. The general rule is: the less power is available for the wireless link the smaller the communication range of the wireless link. Of course there are other issues that can greatly influence the quality of the wireless link such as the type

[1] The term "message passing" refers to the transmission of message packets at the hardware level.

of antenna that is used. Nevertheless, even with the latest state of the art of hardware and the best infrastructure to support wireless connections we can conclude that disconnections occur frequently. An example of this is the quality of conversations over a mobile phone, which are at times problematic even though there are a great number of antennas posted throughout many cities nowadays. Another source for disconnections are caused by the use of a finite power source in a mobile device. When a battery of a mobile device is discharged then the device stops functioning and active connections are lost or wireless links may be manually or automatically turned off to conserve battery power. From this we can conclude that mobile distributed systems are *intermittently* connected as opposed to fixed distributed systems that usually have *permanent* links.

- **Type of Execution Context**: Another facet that is maybe less explicit in the definition above is the execution context of a distributed system. With the term "execution context" we refer to the context information that can influence the behavior of an application. Typically in fixed distributed systems the execution context is more static than with the mobile variants. For example, the quality of a connection can depend on the environment in which mobile devices communicate while the quality of a connection in a fixed distributed system is often continuously stable. Another important type of execution context that is influenced by the location of mobile devices is the availability of services. In mobile distributed systems the availability of services often coincides with the location of the mobile device, whereas in a fixed distributed system services are often continuously available for an application.

Ubiquitous computing scenarios entail that computing technology is embedded in all types of devices, ranging from washing machines and refrigerators to cars, clothes and wrist watches. It is clear however, that most of the cooperation between these devices will occur over wireless communication media. Wireless communication media has one important advantage over fixed communication media. Namely, wireless communication media makes the users oblivious to the computing technology in the face of mobility.

Based on this conceptual framework of distributed systems we can further distinguish between two types of different mobile distributed systems [MCE02]:

- *Nomadic distributed systems* have a mix of fixed and mobile characteristics. A nomadic distributed system is built out of fixed and mobile devices that interact and cooperate via infrastructure. This infrastructure can be composed of wireless access points that are themselves connected via a fixed network. An example of such a distributed system is a mobile phone network, where each phone connects to an antenna and the different antennas are connected via cables. As users move about with their mobile phone the connection is transparently carried over from one antenna to another.

- *Ad-hoc mobile distributed systems* consist of a set of mostly mobile devices that are connected via extremely variable quality links and execute in dynamic environments. For example, mobile devices can be completely isolated from other devices and groups of communicating mobile devices

may spontaneously emerge in the environment. Ad-hoc mobile distributed systems further distinguish themselves from their nomadic variants in that there is no infrastructure that supports the communication between devices. Such a network that emerges due to the mobility of the mobile hosts is often called a *mobile ad-hoc network*.

Both types of mobile distributed systems, discussed above, can be used to realize ubiquitous computing scenarios. For example, nomadic distributed systems can be useful to realize ubiquitous computing scenarios in the context of a restricted environment such as an office space or at home. Nevertheless, the vision of ubiquitous computing is not a delimited concept that starts in a restricted environment and stops when you leave it. For this reason ad-hoc mobile distributed systems are needed to further support the scenarios that continue outside of restricted environments such that no assumptions on the available infrastructure can be made.

2.3 Hardware Phenomena

In this dissertation we focus on the concurrency and distribution aspects that arise when developing a distributed application for distributed systems within the range of the nomadic and ad-hoc variants. To distill the important issues that arise we examine the important phenomena that manifest from the hardware that is used to construct these types of distributed systems.

With the current state of commercial technology, mobile devices are often characterised by having scarcer resources (such as lower CPU speed, smaller memory and limited battery) than traditional hardware. However, we cannot help but notice that in the last couple of years, mobile devices and full-fledged computers like laptops are blending more and more. That is why we do not consider these restrictions as fundamental to software development as we consider the following phenomena to be:

- *Volatile Connections*: Two processes that perform a meaningful task together on two cooperating devices cannot assume a stable connection. The limited communication range of the wireless technology combined with the fact that users can move out of range can result in broken connections at any point in time. However, upon re-establishing a broken connection, users typically expect the task to resume. In other words, they expect the task to be performed in the presence of a volatile connection.

- *Ambient Resources*: If a user moves with his mobile device, remote resources become dynamically (un)available in the environment because the availability and location of a resource may depend on the location of the device. This is in contrast with fixed networks in which references to remote resources are obtained based on the explicit a priori knowledge of the availability of the resource. In the context of mobile networks, the resources are said to be ambient.

- *Autonomy*: Most distributed applications today are developed using the client-server approach. The server often plays the role of a "higher authority" which coordinates interactions between the clients. In mobile

networks, and especially in mobile ad hoc networks, a connection to such a "higher authority" is not always available. Every device acts as an autonomous computing unit.

- *Natural Concurrency*: In theory, distribution and concurrency are two different phenomena. For instance in a client-server setup, a client might explicitly wait for the results of a request to the server in order to resume its computation. But since waiting undermines the autonomy of a device, we conclude that concurrency is a natural phenomenon in software running on mobile networks.

These four hardware phenomena are very important because they are inextricable consequences of the hardware components that are used to build such mobile distributed systems. Hence, we will use these hardware phenomena as our guiding principle for the design of the new paradigm we discuss in the next chapter and for the evaluation of the state of the art discussed further on in this chapter.

2.4 Concurrency and Distribution

Although concurrency and distribution are theoretically not the same, the implementation of a distributed system is almost always concurrent. As a consequence a good concurrency model is the foundation of a model for distribution.

In their book "A Theory of Distributed Objects" [CH05] Caromel and Henrio distilled important aspects to consider for parallel and distributed languages and frameworks. In this section we review these aspects in the light of mobile distributed systems and the subsequent hardware phenomena we have discussed in the previous two sections. In the light of these hardware phenomena we will find that these aspects can no longer be sufficiently addressed with traditional methodologies. This insight is important because it influences design decisions of the distributed languages and middleware that are discussed in the subsequent sections.

2.4.1 Definitions

This section gives definitions of the main concepts that need to be modeled by parallel and distributed languages. The first definition is about the real world. The world consists a number of activities that are occurring at the same time.

Definition 2 (Parallelism) *Execution of several activities or processes at the same time.*

Parallelism is often introduced into a program for two reasons:

1. A first reason for using parallelism in programs is to increase the efficiency of programs. If a certain task can be divided over multiple processors, then that task can be completed faster than when it would run a single processor.

2. Secondly, it can be used as a modeling technique. Sometimes the program needs to deal with inherently parallel concepts. This is sometimes the case

when a program is a simulation of the real world, for example simulations are sometimes made to measure the congestion rate of roads on arterial roads. Another example is when an application has to interface with the real world by means of sensors and actuators. In that case the application needs to respond to events, which are possibly occurring at the same time.

Applications developed for mobile networks will typically fall into the second category. Since such applications typically interact with the physical world, through their wireless and possibly other sensor interfaces, they have to deal with the inherent parallel nature of the ambient in which they run.

The second definition, concurrency, is a consequence of parallelism: when multiple activities manipulate a resource at the same time.

Definition 3 (Concurrency) *Simultaneous access to a physical or logical resource.*

Concurrent access to a resource can render it in an inconsistent state. For example, if two people would simultaneously enter a sentence on the same typewriter then the result will be inconsistent and unreadable and contain a mixture of characters of the two sentences that were entered on the keyboard. Note that the words "inconsistent" and "unreadable" are with respect to the two original sentences that were input by the two people. If the two people were asked to enter a number of independent characters simultaneously onto the typewriter, then the sequence of characters is still readable in spite of the fact that they will be mingled. Hence, what is important is that *consistency of state depends on the semantics of the program.*

Many of the concepts that are introduced in a parallel or distributed languages deal with concurrent access must allow the developer to specify what is consistent state and depending on the semantics the correct level of interleaving accesses of parallel activities has to be enforced.

The next definition is about distribution from a software perspective:

Definition 4 (Distribution) *Multiple address spaces.*

Middleware and distributed languages need to introduce a means to address data that exists in another memory space. Multiple address spaces also means that at the lowest level no data can be shared by parallel activities. This is in contrast to symmetric multiprocessors (SMP) where a number of processors work directly on the same memory. When dealing with multiple address spaces messages have to be exchanged over the network to access data in another address space.

When working with multiple address spaces a distinction can be made between two types of distributed systems:

Definition 5 (Synchronous systems) *A synchronous distributed system has been defined [HT94] as one where the following bounds are known:*

- *the time to execute each step of a process has known lower and upper bounds;*
- *each message transmitted over a channel is received within a known bounded time;*

- *each process has a local clock whose drift rate from real time has a known bound.*

The definition above matches best with a distributed system where complete control can be retained over the hardware that is used in its setup, for example in a local area network. In a local area network computers are typically connected directly to one another, the type of computers and processor speeds that are connected and the type of network connection is known. However, although such a setup would allow one to make good estimations of the different boundaries it would still not be accurate. For example, the workload of the computers and the network usage could cause congestion such that boundaries are eventually exceeded. Another cause for the boundaries to be exceeded is the occurrence of network partitions. Network partitions can prevent messages from arriving within a bounded time. Nevertheless, distributed programs sometimes assume a totally controlled world where the boundaries are known and guaranteed in order to make the software requirements simpler.

Completely opposite to synchronous systems are asynchronous systems:

Definition 6 (Asynchronous systems) *An asynchronous distributed system cannot set any boundaries, hence*

- *there is no boundary on the execution speed of a process;*
- *there is no boundary on the time it takes for a certain message to arrive at its destination;*
- *the drift rate of a local clock is arbitrary.*

As we explained above a synchronous system can sometimes be assumed in certain settings. However this is no longer possible in the context of open networks, such as mobile networks or the Internet. First, in an open network we have to deal with heterogeneous hardware for which the specifications are not known. Hence, various execution speeds are attained. Second, the type of the network implies that no estimations for boundaries can be made. For example, in a mobile network the delivery of messages can be directly linked to the (unpredictable!) mobility of the user carrying the mobile device. Finally, the drift rate of the clocks in the different mobile devices cannot be predicted since synchronization of clocks implies communication which itself is unpredictable.

Note that although distribution and concurrency are typically associated with one another distribution does not always imply concurrency. This can be illustrated through the following example. A client application in a client-server model typically makes a single connection to a server and when it sends requests to the server it waits until these requests are answered. Hence, although the overall distributed system may execute in a concurrent fashion the clients themselves are not necessarily concurrent. While concurrent accesses to shared resources give rise to data consistency issues in a program, distribution gives rise to another set of problems, more particularly the need to deal with independent failures. A failure of a machine or communication link implies that parts of the program continue to run while other parts may become (temporarily) unavailable for communication.

Now that we have discussed the important concepts that are encountered in distributed systems we can discuss how they can be dealt with at the software level.

2.4.2 Denoting Parallel Units in Programming Languages

A first important concept that we find in software is the ability to spawn parallel activities. A parallel activity is expressed as a parallel unit. Such a unit can range from a process to the level of expressions [BST89].

Processes and Threads

Processes are perhaps the most frequently used unit of parallel activity. Many operating systems run each program in a separate process. Processes have their own state and data, hence no memory is shared between processes. Within such a process it is possible to create multiple threads. In contrast to processes, threads can share memory and allow for more fine-grained parallel activity. In most mainstream languages threads are created dynamically and terminate when the top-level procedure they are executing returns. However, often some functionality is provided to abort a thread such that it terminates before this top-level procedure has returned.

Objects

There are several options to introduce parallel activity onto objects. In a sequential object-oriented programming language objects interact via message passing. An object sends a message to itself or another object and waits until the receiving object has processed the message and returns control back to the sender. This is often paired with a value that is returned to the sender. From this metaphor Bal et al. [BST89] distill four different options to map parallelism onto an object:

1. attach a thread to an object and the object can be active without having received a message.
2. allow the object to continue its execution after it has sent the message. In other words, an object sends a message and continues its execution without waiting for the receiver to have completed processing the message.
3. instead of sending the message to a single destination the message can be sent to multiple objects that each process the message in parallel. The sender waits until the different receivers have finished executing the message.
4. the receiving object continues executing after it has returned control to the sender.

Making a choice between these options involves a number of considerations. For example, the first option does not integrate well with the paradigm of object-orientation since objects are no longer solely activated by means of message passing. These and other integration considerations are discussed in section 2.5.2.

Expression and Statements

The most fine-grained unit of parallelism is expressed at the level of expressions and statements. For example, in Occam it is possible to declare the parallel execution of a number of statements using the `PAR` keyword. As noted in

[BST89], parallelism at the level of statements or expressions is easy to understand and use but difficult to maintain in large applications, because it is more fine-grained.

2.4.3 Design Issues in Communication

After having discussed the different ways of introducing parallelism in programming languages we can now turn our attention to the way one logically distributed parallel unit communicates with another by message sending. There are four important characteristics [BST89] that must be considered for the communication between parallel units in the context of distributed systems. These characteristics are further discussed below.

Characteristic #1: Addressing Parallel Units

A first consideration is *how to address* distributed parallel units. Addressing a parallel unit can be either *direct* or *indirect*. A parallel unit is addressed directly when its communication partner addresses it explicitly. An example of this explicit addressing are remote object references, where an object is directly referred to by another object. On the other hand, indirect addressing of a parallel unit occurs when its sender does not refer to it directly, but instead refers to an abstract intermediary communication partner that in its turn refers to the parallel unit that needs to be addressed. Note that indirect addressing offers greater flexibility and a higher level of abstraction that can be useful to deal with the ever-changing environment in which mobile devices are used.

Characteristic #2: Implicit vs. Explicit Communication

Once a parallel unit can be addressed it is possible to communicate with it. A parallel unit sends a message that is received by another parallel unit. Sending a message is usually always explicit in the code, but receiving a message can be either implicit or explicit. For example, when a method is invoked on a remote object in Java RMI, then this method invocation is implicitly accepted. The object does not have to accept the message explicitly. Occam is a language where messages are accepted explicitly. In Occam syntax is provided such that a process can explicitly listen to a channel for a message.

Characteristic #3: Communication Timing

Another important decision is whether to use synchronous or asynchronous communication between parallel units.

Definition 7 (Synchronous Communication) *The sender and the receiver both synchronize at every message.*

Asynchronous communication is often defined from the perspective of the sender [BST89]:

Definition 8 (Asynchronous Communication) *The sender does not wait for the receiver to be ready to accept its message.*

The decision on which type of communication that should be used is not binary, it can be seen as a continuum between synchronous and asynchronous communication. Caromel and Henrio distill a number of these points in the continuum [CH05]:

- **Asynchronous communication with rendez-vous**: The sender of a message blocks until it has received an acknowledgment from the receiver that is has been received. However, the sender does not wait until the message has been processed.
- **Asynchronous communication with FIFO order**: The sender of a message is guaranteed that the messages it sent to a receiver are received in the order it has sent them.
- **Asynchronous communication without order guarantee**: Messages sent by a parallel unit can be received in any order, irrespective of the order in which the messages were sent. An example of a low level messaging protocol with such semantics is UDP.

The latter two types of asynchronous communication match well with inherently asynchronous distributed systems such as the ones found in mobile distributed systems, because they allow one to abstract from unavailable devices such that the autonomous nature of devices is not hampered. This is further discussed in section 3.3.

Characteristic #4:Reliability

There are several degrees of reliability that can be guaranteed when communicating. For example, on the one end when communicating using UDP there are no delivery guarantees made to the sender of a message. If a message is lost in transit it will never arrive at its destination. On the other hand, the sender of a message can continuously retry sending a message until it receives an acknowledgment that the message has been received. Between these two extrema there are approaches that provide some fault tolerance to a limited extent. An example of this is the TCP/IP protocol, which is frequently used for connections over the internet. In any case, there is no single strategy that is suitable for all distributed applications. However, in the case of mobile distributed systems, where volatile connections are the rule rather than the exception care must be taken in making the correct choice. Note that the degree of reliability is independent from the communication timing. On the one hand, synchronous communication can be made reliable by blocking the sender and meanwhile retrying to send the message. On the other hand asynchronous communication can be made reliable by transparently resending a message until an acknowledgment has been received.

2.4.4 Corollaries of Mobile Distribution

Above, we have discussed fundamental concepts found in distributed systems and how these concepts can be translated into programming language concepts. We can now revisit the consequences of these choices in the context of mobile distribution.

Non-Deterministic Interactions

Non-deterministic interactions are a distinct characteristic of distributed systems and are a consequence of the use of multiple independent machines (which have their own internal clock and speed) that are acting on a shared resource. An important insight is that the type of communication determines to a large extend the degree of non-determinism that can occur. For example, suppose a distributed bounded buffer object. The buffer is accessed by a single producer and multiple consumers. This example can be used to compare synchronous communication with and without an explicit receive statement. One problem is what happens if the buffer is empty. In the case of an explicit receive statement in Occam for example, the buffer can execute a statement: `writer?element`. In this case the buffer object will wait explicitly until the writer has sent an element. In the case of a communication model without an explicit receive statement methods of the buffer object are executed in the order they are received. This order is dependent on the internal clock of the consumers and the producer and the quality and speed of the connection, which are variable and cannot be predicted. Hence, the former communication type is more deterministic than the latter. However, when comparing synchronous communication to non-blocking communication we find that the latter form introduces even more non-determinism. The extra degree of non-determinism is caused by the fact that computation is continued immediately after sending a message irrespective of whether the message has been accepted. Non-determinism can be reduced by introducing synchronization mechanisms in the concurrency model such that the program can maintain a consistent state.

Asynchronous communication decouples the sender from the receiver and therefore behaves better with respect to the autonomous nature of devices in a mobile distributed system. In the next chapter we will discuss criteria that reconcile the consequences of this non-determinism with the autonomous nature of devices.

Partial Failures

As explained in sections 2.2 and 2.3, volatile connections are inextricably associated with wireless communication media. Also, mobile devices can fail temporarily due to batteries that are drained. Failures are generally a hard problem in the context of distributed computation. In a distributed system a component (network link or device) can fail while the other components in the system are unaffected and continue their computation, hence the name partial failures. In a distributed system a failure generally cannot be detected accurately. Failures are nowadays most often detected based on timeouts. The problem is that timeouts are only an estimation. Latencies of messages can vary based on the load of the network and the machine, such that a message that is considered to be lost because no reply has been received within a certain time interval could still be processed and return the reply too late. It is also possible that a message has been received by a node, but that the link failed just before a reply can be sent. This makes it generally impossible for a sender to determine whether a message that is considered to be lost has actually been received. As a consequence, when such failures are dealt with by sending messages twice it is possible that messages are received multiple times. It is also generally impossible for the sender

to determine which component has failed. Either the device or the network link could have failed.

Current mainstream distributed models, such as the ones found in CORBA or other remote method invocations schemes deal with such partial failures by propagating exceptions. However, in mobile networks volatile connections are the common rule rather than the exception. As a consequence programming mobile distributed systems in such models is hard.

A number of conceptual solutions have been developed to deal with failures in distributed systems. The most important ones are (distributed) transactions and replication. A transaction guarantees the atomic execution of a set of actions in the face of failures. Atomic execution means that either all actions are serially executed or none of them at all. Another solution to deal with partial failures is replication. Replication is used to ensure the availability of services in a network by duplicating them on multiple machines in the network such that when a machine or network link fails the service remains available on other nodes. Hence, replicated services try to hide network failures. Other techniques are more application specific. For example, when a device coordinating distributed computations fails a new coordinator could be elected. Although these techniques have proven useful in the context of fixed distributed systems, the protocols associated with these techniques generally do not scale to mobile distributed systems. This is mainly because these protocols typically rely on centralized coordination and expect failures to be rare and of short duration. An example is the 2-phase-commit protocol [CDK05] used to support distributed transactions. In the 2-phase commit protocol there is a coordinator that asks all participants in a distributed transaction if they are able to commit the actions associated to the distributed transaction. The participants in the transaction answer "Yes" or "No". If the coordinator receives a "Yes" from all participants then it sends a commit instruction to all participants. If any one of the participants answers "No" then the coordinator sends an abort instruction to all participants to all other participants. The result is that the transaction is either committed as a whole or not at all. Note that this scheme only works because a participant that answered "Yes", cannot change this decision until it receives either a "commit" or "abort" message from the coordinator. Hence, if the connection between the coordinator and the participants fails after a number of participants voted "Yes", then these participants cannot perform any operation that would render its vote invalid. In a fixed distributed system, where failures are exceptional and systems can be closely monitored for failures, such problems can be solved in an acceptable timeframe. This is in contrast to the failures encountered in mobile distributed systems that are due to volatile connections. Volatile connections are common and the time to restore a connection can be directly related to the mobility of a user.

2.5 Objects vs. Concurrency and Distribution

Above we have discussed how concurrency and distribution concepts can generally be addressed in software. We have also discussed the consequences of these choices in the context of mobile distributed systems. Now that we have done this we can turn to a specific paradigm to express concurrency and distribution. The object-oriented programming paradigm provides a good foundation for dealing

with distribution and concurrency, because it successfully aligns encapsulated objects with concurrently running distributed software entities [BGL98]. However, there are a number of different approaches how distribution and concurrency issues can be expressed in the paradigm. Briot et al. [BGL98] make a distinction between the *library*, the *integrative* and the *reflective* approach. These approaches are discussed in the following subsections.

2.5.1 The Library Approach

Distribution and concurrency primitives are encapsulated and are modeled *using* the object-based techniques. Using aggregation and inheritance the primitives can then be integrated in the application. In this approach is that two kinds of objects are used. One kind is used to express the solutions to the issues associated with the concurrency and distribution, while another kind is used to model the domain concepts in the program. Both kinds of objects sometimes need to be mixed to implement the correct solution. An example of this is the `Thread` class found in many libraries for introducing concurrency in an object-oriented language. Concurrent objects, which often implement domain concepts, need to inherit from this class and implement the `run` method. This example illustrates how concurrency and domain concepts are composed together based on the inheritance relationship. The composition of two different kinds of objects generally results in two problems. A first problem with this approach is that the distinction between domain objects and objects that deal with concurrency and distribution issues is obfuscated. A second problem with this approach is that the library, as in the example above, sometimes enforces a structure onto objects that model domain concepts such that modularizing domain concepts can become impossible. A direct consequence of this is that the extensibility of the different kinds of objects becomes more difficult after they have been composed.

2.5.2 The Integrative Approach

In section 2.4.2 we have already discussed different approaches to introduce concurrency into the object model. In this subsection these approaches are considered again from an integration perspective. The integrative approach aims to align concurrency and distribution concepts with the object paradigm. The integration is achieved by merging some of the concurrency and distribution concepts with the concepts found in the object paradigm. This approach alleviates some of the problems found in the library approach. First, since major concurrency and distribution aspects are merged with concepts of the object paradigm the programmer has to deal with less concepts. This enhances the understandability of the concurrency and distribution aspects of the program. Second, there is less need to manage the concurrency and distribution aspects of a program, provided the object paradigm is aligned intuitively with the concurrency and distribution concepts. The three main dimensions along which concepts can be merged are discussed below.

Object and Process

The integration of an object with a process leads to the notion of an *active object*. The two concepts can be unified because both can be regarded [Mey93] as an encapsulated unit that can communicate with others. An object can have none, one or multiple processes associated with it. An object that does not have any process associated with it is sometimes called a *passive object*. The number of processes associated with an object gives rise to different types of object-level concurrency:

- *Serial or atomic:* only one message is computed at a time.
- *Quasi-concurrent:* multiple object activations within an object can exist at a single point in time. Nevertheless, at most one activation can be executing at a time. The other activations must be suspended at that time.
- *Concurrent:* multiple unsuspended activations can be present at a single point in time. However, certain restrictions on the concurrency may exist. These restrictions are necessary to maintain a consistent state.
- *Fully concurrent:* is the same as concurrent objects but without any concurrency restrictions. Fully concurrent object models are functional by nature so that state does not change during a method execution and no inconsistent state can occur.

An important issue with regard to the different types of object-level concurrency is maintaining a consistent state. Quasi-concurrent and concurrent objects are susceptible to race conditions at the level of individual instructions within a method, because concurrent object activations within the same object can result in a non-deterministic interleaving of instructions. On the other hand, serial and fully concurrent object models cannot have race conditions at the level instructions of a method. Nevertheless, as shown by Briot and Yonezawa [BY87] in the case of serial objects race conditions can still occur at the level of interactions. They give the example of a counter object with `set` and `get` methods. Clients want to increment and decrement the counter using these methods. Due to the non-deterministic interleaving of the `get` and `set` methods updates can get lost. Suppose the counter is initialized at zero and two clients want to increment the counter by one. Consider the following schedule: both clients request the state of the counter and in both cases the result returned will be zero. Next, both clients update the state of the counter and set it to the result of the `get` invocation incremented by one. The resulting state of the counter is one. Hence, one counter update can be lost due to the non-deterministic interleaving of messages.

Now that we have discussed the different levels of concurrency that can exist within an object we can turn to how concurrency can be initiated in the object paradigm. There are two approaches objects can be activated: reactive vs. autonomous activation. In the case of reactivity object activation coincides with method invocation. A message is sent to an object and the object is activated by this message. In the case of autonomy an explicit process is associated with a concurrent object. The object starts running from the moment it is created, with little or no regard to external events. The object paradigm naturally

matches better with reactive object activation, but autonomous activation usually gives more fine-grained control over the concurrency issues. For example, autonomous activation offers constructs that allow an object to explicitly receive messages, whereas reactive object models are often based on implicit message acceptance (both were discussed in sections 2.4.3 and 2.4.4). Hence, when integrating processes and objects a choice has to be made whether the active object preserves the reactivity principle or whether an autonomous object system is adopted.

Object Activation and Synchronization

A second type of integration merges the method invocation and process synchronization concepts. Merging both concepts gives rise to the notion of a *synchronized object*. When multiple processes are executing in parallel and working on shared resources there is a need to synchronize parts of a program such that it exhibits the correct semantics and prevent that the concurrent accesses lead to an inconsistent state. There are two levels at which synchronization can be integrated with concepts from the object paradigm:

Message Passing Level Synchronization In a sequential object oriented language the sender of a message waits for the receiver to execute the message and return the result of the method invocation. This same mechanism can be used to introduce synchronization between active objects and is also known as *synchronous message passing*. An active object can send a message to another active object and wait until that object has processed the message and sent back the return value. Message passing forms a natural means to synchronize two concurrently executing objects such that the resulting semantics remains close to sequential semantics. However, in a mobile distributed system, where the latency of messages sent between objects can be high such semantics can harm the autonomous nature of devices. A variant that hides the latency of objects is *asynchronous message passing*. In this case the sending active object does not wait until the message it sent is actually delivered or even processed. An issue that complicates the use of asynchronous message passing are return values. After all, when an active object does not wait until the callee has processed the result it cannot return the result. Typically callbacks are used to process the return values of asynchronous messages, but methods that are used as a callback clutter the code since for each different context in which an asynchronous message is used a callback method needs to be implemented. Another disadvantage of callback methods is that they break the flow of a typical object-oriented program and harm the readability and understandability of the program. To overcome this problem a linguistic abstraction, called *futures* or *promises*, have been proposed [LS88] and implemented in a number of programming languages.

Object Level Synchronization Sometimes more explicit synchronization control is needed that cannot be expressed solely at the message passing level. The necessary degree of control over the synchronization is related to the degree of object-level concurrency:

- *Intra-Object synchronization:* when multiple object activations within one method can be active at a single point in time there is a need to ensure the

consistency of the internal state of the object. Usually, there is a need to specify which methods need to be executed in a mutually exclusive fashion. Note that in a serial active object all methods are mutually exclusive by definition. Although such a serial active object might be considered less expressive, because it restricts the degree of parallelism, it has the benefit that it eliminates inconsistent states that result from concurrent accesses to the internal state of an object.

- *Behavioral synchronization:* it may be possible that an object, depending on its current state, is temporarily unable to perform methods that are part of its interface. A typical example is a queue that when empty cannot execute an enqueue method invocation until a dequeue method is executed.

- *Inter-Object synchronization:* sometimes synchronization is necessary between a set of objects to perform a certain task. An example of such a more global synchronization is that of a distributed transaction where a hierarchy of objects are involved to atomically perform tasks. An example of this type of synchronization is a banking application where one account must be credited while a number of other accounts must be debited atomically. More complex synchronization schemes are needed to achieve such synchronization.

The integrative approach minimizes the number of concepts by integrating and unifying concepts of distribution and concurrency. This approach has the advantage that the aspects of distribution and concurrency are more naturally dealt with and are easier to master. However, the integrative approach lacks adaptability and flexibility of the concurrency and distribution concepts offered by the library approach. In other words, the concurrency and distribution concepts cannot always be adapted to the requirements of the applications.

2.5.3 The Reflective Approach

Thus far we have discussed the library and integrative approach. The reflective approach provides a bridge between both approaches. Briot et al. [BGL98] have noted that the library approach has the advantage that it allows developers to structure distribution and concurrency into reusable concepts that can be modified thanks to the different extensibility and reusability mechanisms offered by object-oriented techniques. This in effect gives a high degree of flexibility which allows the customization of distribution and concurrency to new contexts. A middle ground between both approaches is the reflective approach. The reflective approach can be regarded as a bridge between the library and integrative approach. The idea is to integrate libraries into the programming language via a meta-object protocol (MOP). A MOP allows modifications to the concepts of the object paradigm. In other words, by using the MOP of a language we can unify concurrency and distribution concepts with the language and still have the flexibility offered by the library approach [BGL98].

2.5.4 Discussion

Not much experience exists regarding the development of mobile distributed systems, and where it is even more difficult to foresee the type of applications

that will be developed in the future by means of this hardware constellation, it might be too early to come up with a language that provides the necessary language abstractions immediately out of the box. On the other hand, the integrative approach is more attractive than the library approach because the focus is more on finding the right language abstractions and therefore this approach provides more insight in the nature of how mobile distributed systems are programmed. An integrative approach aims at reducing the number of concepts that need to be mastered by the application developer such that we can focus on finding a minimal set of building blocks. These benefits of the integrative approach correspond well with the goals we have set in section 1.1.3.

These benefits also support our choice, which was discussed section 1.4.2, for a language-based as opposed to a middleware based approach. The language-based approach corresponds best to the integrative approach, because the focus is on concept integration, whereas middleware approaches lean more towards the library approach and do not necessarily consider the integration aspect. However, since the necessary programming abstractions for programming mobile distributed systems have not yet been well established we will opt for a language that can be extended reflectively.

In the remainder of this chapter we will discuss some of the existing distributed technologies in the light of programming mobile distributed systems. In the light of the difference between both approaches we make a distinction between distributed programming languages and middleware technologies.

2.6 Distributed Programming Languages

Distributed programming languages usually fall into the category of the integrative approach, which was described above. Although the integrative approach seems most rewarding with respect to new insights how applications for mobile distributed systems should be developed we have found that, to the best of our knowledge, no distributed language has been designed to specifically deal with the hardware phenomena exhibited by the components to construct mobile distributed systems in section 2.3. We will therefore discuss existing object-oriented distributed programming languages with respect to these phenomena. Existing distributed languages can be categorised as languages designed for local area networks and languages that have been designed for open networks, such as the internet.

Mobile networks are inherently open due to ambient resources that can appear and disappear unheraldedly in the ambient of a device. Therefore languages for open networks might have interesting properties with respect to programming languages for mobile distributed systems. Below we will review two such languages, E and Salsa. Besides these two languages we also review Argus, because it is a language specifically designed to deal with partial failures. In mobile distributed systems partial failures occur frequently due to the volatile connections. However, before we review these languages we first discuss ABCL because it tries to reconcile an object-based concurrency model with mutable state.

```
[object Buffer
  (state declare-the-storage-for-buffer)
  (script
    => [:put aProduct]        ; aProduct is a pattern variable
       (if the-storage-is-full
        then (select               ; waits for a [:get] message
                    (=> [:get]
                       remove-a-product-from-the-storage-and-return-it))
       store-a-Product)
    => [:get]
       (if the-storage-is-empty
        then (select               ; waits for a [:put] message
                     (=> [:put aProduct]
                       send-aProduct-to-the-object-which-sent-[:get]-message))
        else remove-a-product-from-the-storage-and-return-it )) ]
```

Table 2.1: Bounded Buffer in ABCL

2.6.1 Actor Based Concurrent Languages (ABCL)

ABCL is a set of object-based languages [YBS86, TMY94] that were initially designed for parallelism and in a later stage also for distribution. Table 2.1 shows the implementation of a bounded buffer for product items written in ABCL. An object is composed of a list of state variables and a script. The script defines the messages that are accepted by the `Buffer` object. Based on the `select` statement an object can change the messages it accepts and provide a different behavior when it receives a message. For example, when a `get` message is received by the buffer when it is empty the buffer will only accept a `put` message and the behavior for that `put` message is to immediately send the product to the sender of the original `get` message.

In ABCL objects are active entities that can be in one of the following states:

- dormant: this is the case when no method activations in the active object are present. This state occurs when the object has processed all the messages in its queue.
- waiting: an object can be in a waiting state when it is waiting for a particular set of messages to arrive. For example, when the buffer is empty and the queue object receives a `get` message then the object has to wait until it receives a `put` message before it can proceed with the `get` message. To specify such an explicit wait ABCL introduced the `select` statement which allows the specification of the messages an object is waiting for.
- busy: an object in a busy state when it is processing a message. In ABCL only one message can be processed at a time. However the processing of a message may be interrupted based on the express message passing mechanism, discussed below.

In ABCL different types of message passing mechanisms are introduced from which a developer can choose. There are two aspects of message sending in ABCL. The first aspect is the delivery mode of the message. A distinction is made between *ordinary* and *express* mode messages. Ordinary mode messages can be preempted by express messages. When an ordinary message is being processed and an express message arrives then that process is preempted and the express message is processed. When an express message is processed and one or more express messages arrive then these express messages are processed in

the order of arrival. When no express messages remain to be processed then the ordinary messages are further processed in the order of arrival. Hence, ABCL features *quasi-concurrent* active objects. The second aspect is the message type that needs to be chosen:

- "past" type messages are sent without waiting for the message to be processed by the receiver. This type of message passing does not return a value.
- "now" type messages are sent and the receiver explicitly waits until the message is processed. A value is returned to the sender of the message after it is processed.
- "future" type messages are a combination of past type and now type messages. This type of message passing immediately, that is without waiting for the message to be processed, returns a "future" value. Such a "future" value can be queried for the result and if the result is available, then the result is returned, otherwise the active object that queried for the result is blocked until the result becomes available.

The "now" and "future" type message passing can be reduced to "past" type message passing combined with the use of a `select` statement. The "now" type message passing is based on the select statement. After the sender sent a "now" type message it invokes the `select` statement to receive the reply message such that the object ends up in a waiting state. When the receiver has processed the message and returns the result this result is sent back to the sender such that the sender will return back to a busy state and can process the result. "Future" type message passing is reduced to "past" type by means of a future object that is passed along with the message to the receiver. The receiver sends the result to the future object that is queried by the sender. The sender can query the future object for the result with the `next-value` operation. When the future object has not yet received the result the sender blocks until the result has been received. In the other case the result is immediately returned.

Evaluation

Evaluating ABCL/1 with respect to the hardware phenomena we discussed in section 2.3 we obtain the following results.

Volatile Connections The ABCL language family was foremost conceived for supporting parallelism. Although some ABCL languages support remote invocations their behavior with respect to failures is not defined to the best of our knowledge.

Ambient Resources Since the focus of ABCL was to support parallelism no mechanism to support ambient resources is introduced.

Natural Concurrency In ABCL each object is associated with a process and objects process messages in the context of that process as explained above. A consistent state is maintained through the use of the `select` statement and the definition of `atomic` blocks. Code in an atomic block cannot be interrupted by any type of message.

Autonomy ABCL languages do not have any concepts that inherently require a client-server setup. However, the `select` statement which brings an object into a waiting state can result in dependencies between objects. This is especially the case when considering now and future type messages. A now type message results in an object waiting explicitly for an object to return its value. Hence, if a connection breaks while an object is waiting for a result then that object has two options. It can wait until the connection is restored or it can consider the remote invocation as failed. In the former case the object cannot process any other messages while it is waiting for the connection to be restored, thereby harming the autonomy. The latter case results in very complex software, because with each remote method invocation one has to assume that the invocation can fail. Note that dependencies increase as the call graph grows. Future type message passing suffers from similar problems when a future is queried for its result when it has not yet been resolved.

2.6.2 Argus

Argus [Lis92, LS88] is a distributed programming language designed to cope with failures of computer nodes and the network. These failures were captured by augmenting the sequential language CLU with distributed objects that support transactional semantics. In Argus a distinction is made between local objects and *guardian* objects. Guardian objects are remote objects that have one or more processes and can be remotely created and moved around. Guardian objects are interacted with by invoking *handler operations*, which are similar to methods. Handler invocations are served by the guardian object's internal processes. The state of a guardian object is composed of a set of local objects and references to guardian objects. These objects can be annotated with the keyword `stable` to specify that they are recoverable. Such objects are recovered from stable storage after a crash of a computer node. Other objects, which were not annotated with the `stable` keyword typically contain volatile data that can be reconstructed from stable objects or that can be discarded.

Argus also allows atomic data types to be specified. Objects instantiated from such atomic data types, henceforth called atomic objects, differ from other objects in the way operations are handled. Each operation on an atomic object is accompanied by a lock. A distinction is made between read and write locks. Multiple processes can have a read lock at the same time on an atomic object, but a write lock is exclusive. An exclusive lock ensures that no other processes can have a (read or write) lock at that time. The distinction between these two types of locks has the advantage that more parallelism is allowed on the resources while the consistency of the state is maintained. Another difference between atomic and regular objects is that operation invocations on atomic objects can have transactional semantics. This is the case when the operation invocation is annotated by the `action` keyword. In that case state changes are done on a copy of the state of an atomic object. Also, other operations invoked in the control flow of such an invocation may have (nested) transactional semantics. It is only when control is successfully returned to the top level (that is, no aborting exceptions or returns have occurred in the control flow) that all changes made to the objects are committed. The use of such atomic actions is illustrated with the following code:

```
% for a transfer it does the following
```

```
% find out accounts and amounts from user and store in local variables to, from and amt
enter topaction % start a new transaction
  t: branch := get-branch(to)
  f: branch := get-branch(from)
  coenter % start a nested transaction
    action f.withdraw(from, amt)
    action t.deposit(to, amt)
  end except others: abort exit problem end % all exceptions cause abort of topaction
end % topaction
except when problem: % tell user that transfer failed
end % except
% tell user that transfer succeeded
```

This is the code of a transactional bank transfer in a distributed bank application where an amount is withdrawn from one account and deposited on another one, where both accounts may reside on a different branch. The transactional semantics ensure that either both operations are completed successfully or no changes are made at all to the accounts.

Argus supports both synchronous and asynchronous remote method invocations. Asynchronous remote method invocations are introduced into the language through the introduction of a port concept. Each port corresponds to an operation that can be remotely invoked. A receiving entity defines a number of ports, which are associated with an operation that can be called by a client. Ports can be grouped together such that calls to ports in the same group are sequenced. A stream is associated per client object that interacts with such a port group. A stream has several purposes. First, it ensures that messages arrive in the correct sequencing order. Second, it ensures that messages are executed at most once. Finally, a stream is also buffered such that multiple messages can be sent without waiting for them to be processed. Hence, streams enable asynchronous method invocations in Argus.

Argus introduces *promises* as a programming abstraction to deal with return values in the context of asynchronous method invocations. A promise is a built-in programming abstraction that serves as a proxy for the value that will be returned as a result of the asynchronous remote method invocation in the future. A promise is initially in a *blocked* state when it is returned as the result of an asynchronous method invocation. Once, the result is computed the promise is *resolved* with this result and the promise *ready* state. The promise remains in this state once it is resolved and it also never changes this result afterwards. The result that is represented by this promise has to be retrieved explicitly by means of the `claim` operation. If the state of a promise is *blocked* when the `claim` operation is performed then the caller is blocked until the promise is resolved with the result or when the promise is resolved with an exception. In the case of a *ready* state the result is returned immediately.

Evaluation

Evaluating Argus with respect to the hardware phenomena we discussed in section 2.3 we obtain the following results.

Volatile Connections Argus provides several abstractions to deal with communication failures. First, there are atomic actions which provide an all-or-nothing semantics for remote method invocations in the presence of failures. As explained in section 2.4.4 the underlying two phase commit protocol to support such atomic actions is not feasible in a context of volatile connections, because

communication failures are common and are not guaranteed to be resolved in a reasonable amount of time.

On a lower level, the notion of streams also introduces some resilience to failures. In the case of a failure the system will retry to deliver the messages placed on a stream with the guarantee that a message is executed at most once. However, at a not further specified point in time the system will give up and break the stream. A distinction is made between synchronous and asynchronous breaks. In the former case the connection is broken between two method invocations, whereas in the latter case the connection is broken at the moment a message was being processed. It is important to make the distinction between both type of broken connections because the former implies that all messages prior to the broken connection have been properly handled. In the latter case the system remains in a non-deterministic state because it is generally impossible for the sender to determine the current state of the receiver.

Ambient Resources Argus was conceived in the context of managed networks where network nodes do not need to be discovered because they are added manually. As such Argus does not provide for a means to discover new acquaintances in the network.

Natural Concurrency Argus's concurrency model is based on fine-grained locking mechanisms that make a distinction between read and write locks. This distinction allows more parallelism than locking mechanisms that do not make the distinction, while maintaining a consistent state. However, the distinction between both types of locks is also a cause for deadlocks. Preventing these deadlocks forces one to meticulously determine for all operations whether their order could induce a possible (distributed) deadlock. Distributed deadlocks are extremely hard to resolve in mobile distributed systems since not all parties are necessarily available for communication.

Autonomy Argus was intended to support mainly reliable client-server applications such as illustrated by the banking application. In the case of atomic actions the nodes are dependent on the coordinator of the transaction as a result of the two phase commit protocol. What is more, atomic actions can be nested such that all nodes in the control flow of a transaction become dependent on the coordinator of the transaction. The synchronous and asynchronous remote method invocations suffer the same problems as now and future type message passing found in the language ABCL.

2.6.3 E

E [MTS05] is an object-oriented language inspired by the actor model of concurrency. The language was designed for secure peer-to-peer scripting in open networks such as the internet. In E the unit of concurrency and distribution is a *vat*. A vat conceptually consists of a thread of control, an input queue and a state consisting of objects. Each of these objects can be referenced from objects allocated in other vats. Hence, E's object model does not differentiate between the representation of local and distributed objects. Instead it differentiates in the type of references. Objects allocated in the same vat refer to one another by

means of *near references*, whereas *eventual references* are used to refer objects that were allocated in another vat. The type of reference defines what type invocations are supported. A near reference can carry both synchronous and asynchronous method invocations, whereas an eventual reference can carry only asynchronous method invocations. Asynchronous method invocations arrive in the input queue of a vat and are processed one at a time such that no race conditions can occur.

E introduces a programming abstraction, called promises, to deal with the results from asynchronous method invocations. These promises are inspired by the promises found in the language Argus, but differ from them in a profound manner. Retrieving the result of a promise occurs through the use of the `when` construct, as illustrated by the following example [Mil04]:

```
def temperaturePromise := carPromise <- getEngineTemperature()
when (temperaturePromise) -> done(temperature) {
    println(‘The temperature of the car engine is: $temperature‘)
} catch e {
    println(‘Could not get engine temperature, error: $e‘)
}
println("execution of the when-catch waits for resolution of the " +
        "promise, but the program moves on immediately to this println")
```

The example above shows how the method `getEngineTemperature` is asynchronously invoked (using the `<-` operator) on `carPromise` and how the resulting promise is assigned to the variable `temperaturePromise`. The `when` construct is subsequently used to retrieve the actual temperature from the promise. This construct differs from the `claim` construct found in Argus in that the `when` construct will not stop the control flow should the promise be unresolved. The control flow immediately moves, irrespective of whether the promise has already been resolved. Afterwards, when the promise is resolved with the result the code block is scheduled for execution in the vat's queue and `temperature` will be bound to the result of the asynchronous method invocation. Another difference with promises as found in Argus is that methods can be invoked asynchronously on promises. This is called promise pipelining. The result of this operation is another promise that is eventually resolved after the receiving promise is resolved.

Evaluation

Evaluating E with respect to the hardware phenomena we discussed in section 2.3 we obtain the following results.

Volatile Connections A broken connection breaks the eventual reference and all subsequent invocations on this reference result in exceptions. Hence, broken connections are handled as exceptions. As a consequence E does not support volatile connections because they have to be dealt with as exceptions.

Ambient Resources Although E was conceived as a distributed language for open networks. E does not explicitly support mobile distributed systems and does not offer facilities to discover ambient resources.

Natural Concurrency E's concurrency model is partially based on the actor model, which is discussed in detail in chapter 4. State consistency is preserved, because within a vat a single thread processes messages for all objects such that race conditions on the internal state of an object cannot exist. Further synchronization between asynchronous method invocations is achieved through the reification of promises, which can be manually resolved.

Autonomy E was conceived as a scripting language for peer-to-peer interactions between nodes in an open network. As a result the underlying protocols of the language were designed specifically so that no client-server setup is assumed. Also, the careful design of the promises found in the language ensure that the abstractions which deal with the results of asynchronous method invocations do not create dependencies between vats that could harm the autonomy.

2.6.4 Salsa

The language Salsa [VA01] is a contemporary language that was designed to support dynamically reconfigurable networks such as Internet and mobile computing environments. Salsa supports these networks by means of universal names, mobile active objects and abstractions to support the coordination of interactions. All interactions between active objects occur via asynchronous message passing and their coordination is supported by means of linguistic abstractions for message-based continuations. More particularly the language offers support for token-passing continuations, join continuations and first-class continuations. All of these abstractions are illustrated in the following example:

```
behavior Fibonacci {
    int n;
    Fibonacci(int n)  { this.n = n; }
    int add(int numbers[]) { return numbers[0] +numbers[1]; }
    int compute() {
          if (n == 0)  return 0;
          else if (n <= 2) return 1;
          else {
            Fibonacci fib1 = new Fibonacci(n-1);
            Fibonacci fib2 = new Fibonacci(n-2);
            join(fib1<-compute(), fib2<-compute())
              @ add @ currentContinuation;
          }
    }

    void act(String args[]) {
          n = Integer.parseInt(args[0]);
          compute() @ standardOutput<-println;
    }
}
```

The example above calculates the fibonacci recursively through asynchronous message passing. A continuation is specified by the "@"-symbol. Hence, in the code above the continuation of both asynchronous invocations `fib1<-compute()` and `fib2<-compute()` is the asynchronous invocation `add`, which is automatically passed the combined results of both invocations. Finally, the continuation of the asynchronous invocation `add` is the current continuation of the method in which the computation takes place. In this case it is continuation of the method `compute`. The first time `compute` is invoked asynchronously (in the `act` method) `currentContinuation` will refer to the `println` on the standard

output. Subsequent recursive `compute` invocations will have their current continuation referring to the continuation that computes the previous recursive asynchronous invocation and so forth until the recursion ends.

Evaluation

Evaluating Salsa with respect to the hardware phenomena we discussed in section 2.3 we obtain the following results.

Volatile Connections Broken connections are not explicitly considered in Salsa. However, Salsa is based on the actor model which assumes that messages are eventually delivered. The assumption of eventual delivery is unattainable in the case of mobile networks, because the location of the devices can determine whether a connection will ever be restored. Hence, the failure model does not consider fatal disconnections. Nevertheless, the model of eventual delivery allows one to hide broken connections such that they are not treated as exceptions.

Ambient Resources Salsa introduces the concept of universal actor names. These names are maintained and associated with a naming server. Naming servers have two purposes. First, they are used to retrieve a remote reference to an active object. Secondly, they are used to retrieve the logical location of an actor. Nevertheless, naming servers cannot be used to detect ambient resources.

Natural Concurrency In Salsa an active object processes one message at a time. As a consequence no race conditions on the internal state of an active object can occur. Through the linguistic abstractions for continuation-style message passing different coordination patterns can be expressed. However, to the best of our knowledge behavioral synchronization cannot be expressed in the language.

Autonomy In Salsa, similar to E, all remote communication is asynchronous. The different continuation passing styles support the coordination problems that result from them and prevent that dependencies between active objects occur as a result asynchronous message passing. Hence, the communication mechanisms featured in Salsa do not harm the autonomy of devices in the presence of volatile connections.

2.6.5 nesC

nesC [GLvB+03] is a component-based dialect of C that is designed for programming wireless sensor networks. Wireless sensor networks are composed of a group of sensor-nodes which communicate via a wireless communication link. Each sensor-node collects certain information in its environment and this information is aggregated at certain nodes in the network.

nesC features an event-based concurrency model based on *tasks* and *events*. Tasks are asynchronously executed and do not preempt one another. For this reason it is important that tasks are not computationally intensive and should be short. nesC introduces the keyword `post` to schedule a task for execution.

```
module SurgeM {
  provides interface StdControl;
  uses interface ADC;
  uses interface Timer;
  uses interface Send;
}
implementation {
  bool busy;
  norace uint16_t sensorReading;

  task void sendData() {
    // send sensorReading
    adcPacket.data = sensorReading;
    call Send.send(
           &adcPacket, sizeof adcPacket.data);
    return SUCCESS;
  }
```

```
  event result_t Timer.fired() {
    bool localBusy;
    atomic {
      localBusy = busy;
      busy = TRUE;
    }
    if (!localBusy)
      call ADC.getData();
      return SUCCESS;
  }

  event result_t
  ADC.dataReady(uint16_t data) {
    sensorReading = data;
    post sendData();
    return SUCCESS;
  }
...
}
```

Table 2.2: Component Sampling Sensor Readings and Sending Results

Events are also asynchronously executed, using the `signal` keyword, but can preempt other events or tasks. Race conditions that result from the concurrency can be addressed with `atomic` blocks.

A nesC program consists of component definitions (consisting of interfaces and their implementations) and component configurations. The language features bidirectional interfaces: consisting of *commands* and *events/tasks*. These elements of the interface are to be linked to functions. A component that implements the interface provides an implementation for the commands. Components using the interface provide an implementation for the events or tasks. The events or task implementations are actually callback functions for asynchronously returned results (either as a task or an event). An example component implementation is shown in table 2.2.

The issue of limited resources of the different sensor nodes is addressed in nesC by prohibiting many dynamic features such as late binding and dynamic resource allocation. This restriction has the advantage that compile-time analysis can be performed to detect concurrency issues and code optimization can be performed by inlining functions. The disadvantage of this restriction is that the software cannot be easily adapted to cope with changes in the environment.

Evaluation

Evaluating nesC with respect to the hardware phenomena we discussed in section 2.3 we obtain the following results.

Volatile Connections nesC offers a number of different low-level communication components. These components deal with broken connections as exceptions. As a result the developer always has to deal with them explicitly in the code.

Ambient Resources A broadcasting component allows sensor nodes to detect other sensor nodes. The broadcasting component can be used to detect

Language	Volatile Connections	Ambient Resources	Autonomy	Natural Concurrency
ABCL	∅	∅	∅	√
Argus	by exception	∅	∅	√
E	by exception	∅	√	√
Salsa	Limited	∅	√	No behavioral sync.
nesC	by exception	Partial	Partial	√

Table 2.3: Summary: Evaluation of Distributed Languages

ambient resources in the environment. However, no immediate support is offered to detect the disappearance from resources in the environment.

Natural Concurrency The concurrency model of nesC is based on events. Similar to ABCL there is made a distinction between two types of asynchronous events: pre-empted and non-preempted events[2]. nesC has the advantage that it can offer compile-time analysis of programs to detect possible inconsistencies that occur from this concurrency. Based on the concurrency primitives introduced in nesC high-level concurrency abstractions can be implemented.

Autonomy All remote communication asynchronously returns a result thanks to the use of events and tasks to perform low-level communication between components. However, the concurrency primitives can imply a blocking operation such that dependencies between devices can be created as a result of a blocking operation in an `atomic` block. Nevertheless, the default mode of operation does not imply blocking calls.

2.6.6 Summary

In this section we have discussed five distributed programming languages and evaluated them with respect to the hardware characteristics we distilled in section 2.3. Table 2.3 shows the summary of these evaluations. None of these fully support ambient resources or volatile connections. nesC has the best overall support for dealing with the different hardware phenomena. nesC offers a high-level component-based composition mechanism but unfortunately the distributed properties of the language are low-level. For example, there is no high-level mechanism such as remote method invocations to invoke services on remote components. The high-level languages designed for open networks do not harm the autonomy of devices. In chapter 3 we shall see that this is a consequence of certain characteristics of their concurrency model.

2.7 Middleware

An alternative to distributed languages is middleware. Over the past few years a lot of research [MCE02] has been conducted in middleware for nomadic and ad hoc distributed systems. This bulk of research can be categorized into several groups. In this section we discuss the properties of each middleware category with respect to the hardware phenomena we distilled in section 2.3.

[2] ABCL introduces ordinary and express messages.

2.7.1 RPC-Based Middleware

Alice [HCC99] and DOLMEN [RB99] are attempts to make CORBA feasible for supporting nomadic distributed systems. These attempts focus mainly on making heavyweight ORBs suitable for the lightweight devices and on improving the resilience of the IIOP protocol to failing communication. Other approaches adapt the RPC protocol by supporting queuing of RPCs [JTK97] or enabling rebinding of resources [SBBK95]. These approaches work well when connections are lost for a short time, but do not address disconnections over longer periods of time.

Alice

Alice [HCC99] is an extension of the CORBA architecture. The middleware focusses on low level issues such as address translation support when a mobile host is used from one subnet to another and the IP address needs to change. Some of these issues have already been resolved in the new IPv6 standard, but Alice was made to support backwards compatibility with the IPv4 protocol. Furthermore, Alice does not support autonomous devices because it assumes an infrastructure that provides *mobility gateways* at certain locations in the mobile network. Hence, Alice cannot be used to construct ad-hoc mobile distributed systems. Furthermore, a distinction between clients and servers is made. Nevertheless, both servers as well as clients can run on mobile devices. Volatile connections are supported by the clients, which try to reconnect until a succesful connection can be made. During that time the client is blocked, such that it becomes unresponsive to other events. Mobility of devices is supported through a *handoff* procedure. Such a handoff procedure, which is requested by the mobile host to the mobility gateway, involves creating a tunnel from the previous mobility gateway to the new mobility gateway so that requests and replies are properly forwarded to and from the servers the client was interacting with. Mobility of servers is supported through a mechanism called *swizzling* of the object references. The difficulty of supporting mobile servers in IPv4 lies in dealing with the new network address that is associated with the mobility. There, a mechanism of callbacks is used between the mobile host and the mobility gateways. Mobile hosts that run a server application register a callback to the mobility layer and are notified when their mobility gateway address has changed. At that point the old mobility gateway is contacted such that the address associated with the object reference of the server gets changed to the new mobility gateway. The process of swizzling is similar to forwarding addresses that are used to support the logical mobility of objects in languages such as Emerald [JLHB88].

Rover

Rover [JdLT+95] makes a distinction between *mobile-transparent* and *mobile-aware* applications. Mobile-transparent applications try to hide the consequences of the mobility for the application, whereas mobile-aware applications do not hide mobility for the program such that it can intervene appropriately. Rover makes a clear distinction between clients and servers. Clients run on mobile devices, while servers run on stationary devices. Clients can communicate with servers, but clients cannot interact independently with one another. This harms the autonomous nature of devices and the reliance on infrastructure

makes it only suitable for building nomadic distributed systems. Communication between objects occurs through one of the following mechanisms:

- Relocatable dynamic objects (RDO) are objects which can be dynamically loaded onto a client from the server or vice-versa. The advantage of RDOs is that latencies of the network can be hidden and that in the event of a disconnection the object is still available. An RDO always resides on a server that maintains the primary copy. When the RDO is loaded onto a client then a secondary copy is made. Finally, this copy can be tentatively exported back to the server where the updates of that object need to be reconciled with the primary copy. RDOs also serve the purpose of relocating computation to another device. For example, if a mobile device has to perform a resource intensive operation it can export its RDO to a server and have the computation performed there.

- Queued Remote Procedure Calls (QRPC) enables non-blocking remote procedure calls between RDOs that reside on a client and a server. Thus direct communication is only possible between a mobile and stationary device. The queued remote procedure calls are directly inspired on the stream based approach found in Argus.

Rover differs from other RPC based middleware because it approaches the disconnectedness that results from volatile connections by means of replication. As such, a replicated object can reside on the mobile device and applications can interact with the object without the need for remote communication. However, as explained in section 2.4.4, traditional replication strategies do not scale in the context of volatile connections. The Rover middleware does not define how the replication of RDOs should be handled such that one has to devise an application-specific replication strategy for each object. However, support for this is offered by logging the operations on objects. Section 2.7.4 discusses other middleware to support replication in the presence of volatile connections.

Java Intelligent Network Infrastructure (JINI)

Jini [Wal01] is a service lookup mechanism that was especially designed for ad-hoc networks. In Jini an ad-hoc network consists of a number of Jini lookup services, services and clients. Services register themselves with the Jini lookup service and client find these services by contacting the Jini lookup service. The Jini lookup service differs regular naming services in that it can be automatically found in a network. That is clients and services need not be configured with the network address of the Jini lookup service. Instead they can automatically discover the service in the network. Another difference with regular naming services is that services can be found based on the type of a service rather than the name of a service. In Jini the type of a service is identified with a Java interface. Communication with Jini services is based on the paradigm of synchronous remote method invocations. However, the Jini lookup service provides the stubs to the clients of a service. If the client does not already have a stub to access the service then it is automatically downloaded onto the client. The advantage of this mechanism is that the underlying remote method invocation protocol can be customized based on the properties of the client.

Jini introduces the concept of *leasing* to deal with unanticipated disappearance of devices in the network. Leasing means that relationships between entities in the network have a predetermined finite duration. Hence, instead of creating a relationship between entities in the network until their connection is broken the entities have to agree on a finite duration by which they guarantee to sustain the relationship. After the lease expires the resources associated with the relationship can be cleaned up. Before the lease expires it is possible for the two entities to agree to extend the lease for a new finite duration.

Evaluation

Evaluating RPC-based middleware with respect to the hardware phenomena we discussed in section 2.3 we obtain the following results.

Volatile Connections Most RPC based middleware that was designed to support mobile computing tries to support volatile connections through buffered RPC methods. This approach is similar to the buffered streams found in Argus. Such a mode of communication is resilient to connections that are lost for a short time, but do not address broken connections over longer periods of time. Rover differs from other approaches in that it supports broken connections over longer periods of time because it enables offline QRPC requests to be placed in a first-class log. This log can be manipulated based on the needs of the application. Besides the logging of QRPC requests Rover RDO mechanism allows offline objects to be available on a mobile device such that disconnected mobile devices can continue to function. These mechanisms allow applications to deal with volatile connections. The concept of leases introduced in Jini resolves the important issue that resources are held for an indefinite amount of time but do not solve the fact that failed communication is handled by means of exceptions.

Ambient Resources Of the RPC based middleware solutions Jini is the only technology that allows the detection of ambient resources in the environment. Disappearing resources are detected by means of expired leases.

Natural Concurrency In general no explicit constructs were described to deal with the concurrency that results from mobile devices interacting with one another. Most RPC based models usually have a concurrency model based on explicit threads. However, the RDO communication mechanism offered by the Rover toolkit takes into account that conflicts can occur when a secondary copy is copied back to the server. These conflicts result from the inherent concurrency: two mobile devices can independently interact with their copy of an RDO object such that the interactions lead to conflicts. Rover offers a framework such that the application can resolve these conflicts manually based on the logs of the operations.

Autonomy Many of the RPC-based middleware solutions, such as Alice and Rover are based on a client-server structure such that mobile devices are often dependent on an infrastructure of servers. For example, in Rover all interactions between devices must occur through a server such that mobile devices cannot

directly interact with one another. What is more, the queued RPC requests create dependencies between devices such that the autonomy of devices can be harmed in the face of volatile connections. Unfortunately the setup of Jini is such that each ad-hoc network needs to have a dedicated lookup service in the ambient which can harm the autonomy of devices.

2.7.2 Publish-Subscribe Middleware

A more recent branch of middleware for mobile computing is based on the publish-subscribe paradigm [EFGK03]. The publish-subscribe model aims to decouple the different components in a network. A component can subscribe itself to a remote dispatcher such that published events are delivered to the component. The type of published events the component receives depends on its subscription parameters. The type of these parameters depend on the publish subscribe system and can be based on the type of the event that is published or more advanced filter patterns.

Much of the research effort has been based on making them in large networks. Scalability has been achieved by structuring dispatchers in a tree based setup. Published events are then routed based on this tree to deliver the events to their subscribers. The problem with this approach is that a tree based structure for the dispatchers does not scale in the face of unanticipated mobility, because their topology is static. In other words, the approaches do not support sufficient reconfigurability of the topology of the network. Current research [CJ02, CCW03, CNP00, MC02] investigates other mechanisms for structuring the dispatchers such to support a reconfigurable topology without loosing scalability [CMP05]. Another problem is that although the interaction model of publish subscribe systems decouples the components middleware the programming model requires callbacks to handle results, which can clutter the code and make programs less understandable.

Evaluation

Evaluating publish/subscribe based middleware with respect to the hardware phenomena we discussed in section 2.3 we obtain the following results.

Volatile Connections Traditional publish/subscribe systems offer a variety of message delivery semantics that can deal with failing connections. However, to the best of our knowledge these semantics have not yet been thoroughly investigated in the context of volatile connections.

Ambient Resources Some publish subscribe systems, such as STEAM [MC02], offer abstractions to discover ambient resources. In STEAM an abstraction based on proximity groups [KCM+01]. This abstraction allows components to discover one another when they are located in the same geographical location.

Natural Concurrency In publish/subscribe systems concurrency results from broadcasted events that are received by multiple components. These components process these events concurrently. Hence, concurrency results from a single component that publishes new content that is subsequently concurrently

processed by the different components. Nevertheless, depending on the publish/subscribe system itself other, more concurrent, approaches are possible. For example, an individual component could process different events concurrently.

Autonomy Publish/subscribe systems decouple interactions of components by means of a dispatcher. The decoupling of interactions is positive in the face of volatile connections because dependencies between interacting devices are avoided. Nevertheless, the need for a dispatching service can create dependencies between devices. Most approaches are based on a dispatcher organization such that a device actually relies on that organization, potentially harming its autonomy. However, in STEAM [MC02] this restriction is alleviated by subscribing to event types rather than event systems. This entails that all events are received from all applications in the proximity. However, events are only consumed when they match the event type. Hence, STEAM supports the autonomous nature of devices.

2.7.3 Tuple Space Based Middleware

In the past few years middleware has been proposed [MPR01, DFWB98, MZ04, FMDE04] for mobile computing that is based on Linda tuple spaces [Gel85]. A tuple space acts as an intermediate data structure in which processes can publish and query tuples to communicate asynchronously with one another. Most research on tuple spaces for mobile computing consists of distributing the tuple space over a set of devices. Although tuple spaces are an interesting communication paradigm for mobile computing, the paradigm does not integrate well with the object-oriented paradigm because communication is achieved by putting and querying data in a tuple-space as opposed to sending messages to objects.

Linda in a Mobile Environment

Linda in mobile environments (Lime) [MPR01] is middleware to support both logical and physical mobility of agents and hosts respectively. Each agent is equipped with its own *interface tuple space* (ITS). Access to the ITS is based on the primitives introduced by Linda. Tuples can be added and selected from the tuple space by the agents using the operations `out` and `in`. Furthermore, an operation `rd` is provided to read the tuples from the tuple space. The `in` and `rd` operations are blocking, that is to say that the process will wait until a tuple that matches a pattern is found. Pattern matching is based on templates that contain actual and formal parameters. Actual parameters contain values, whereas formal parameters act as wild cards when they are unbound. Lime also provides non-blocking variants of the `in` and `rd` statements. In the case tuple space contains a matching tuple then that tuple is returned. In the other case no tuple is returned. These non-blocking variants prevent that dependencies between distributed applications are created. However, to prevent such dependencies the application is required to poll the tuple space frequently such that resources are waisted. To avoid the need for polling an application can also register reactions. These reactions are discussed in the last paragraph.

The individual ITSes of each agent are transiently shared with the ITSes of other mobile devices that are currently in the communication range. Sharing

of ITSes occurs transparently and agents can access the contents of all tuple spaces by querying their ITS. Transient sharing of tuple spaces is supported by an atomic join and disjoin operation that is automatically performed by the Lime middleware. A join is performed in the case an ITS becomes available in the communication range of other ITSes. A disjoin operation is performed when a mobile device disappears from the ambient of a device (i.e. when it is no longer in the communication range). The contents of the tuple spaces that were hosted by the device that disappeared are no longer perceived as part of the remaining agents their ITSes after the disjoin operations completed.

Lime also introduces the notion of agent locations, which are addresses for each individual agent that owns an ITS. A tuple is marked with two such locations, a current location and a destination location. The current location points to the address of the agent that added the tuple its ITS. The destination location determines the final destination of a tuple. When a destination location is added to an `out` operation then the corresponding tuple will be atomically moved from the ITS of the agent that added the tuple to the ITS of the agent whose location was given as an argument. This happens when the two devices are in one another's communication range. The operations `in` and `rd` take the current and destination location as additional parameters such that they can be used in the matching process of tuples in the space.

Agents can react to changes in their context by subscribing *reactions* to the ITS. A reaction is composed of an action and a pattern. The action is executed when the corresponding pattern is found in the tuple space. After each operation on the tuple space a reaction that has a matching pattern is selected non-deterministically and this process repeats until no more reactions with a matching pattern exist.

Tuples on the Air

Tuples on the air (TOTA) middleware [MZ04] has a different approach to distribute the tuple space over multiple nodes. As opposed to Lime, individual tuple spaces are not joined into a shared data structure. Instead tuples are empowered with the ability to autonomously move from one tuple space to another. A tuple can only move to neighboring tuple spaces. Two tuple spaces are neighbors of one another when the two devices that host them are in the communication range of one another. Hence, a tuple space can have multiple neighbors depending on the devices in communication range of a device. Each tuple has a propagation rule that is used to decide if a tuple should propagate to its neighbors. At the moment a tuple arrives at the tuple space it can decide to enter the tuple space. If it decides to enter the tuple space then the tuple is stored and the propagation rule is triggered. At the moment a tuple enters it is also given the opportunity to subscribe to events in its current tuple space. Through this mechanism a tuple can react to events after it has entered the tuple space.

Each mobile device runs its own local tuple space and applications can insert propagating tuples in the tuple space. Furthermore, an agent can also subscribe events to its local tuple space such that it can react to changes made to the tuple space by other components of the application and tuples that propagated to the agent's local tuple space. Based on the subscription of such events together with the insertion of autonomously propagating tuples, an agent can launch

queries over a peer-to-peer mobile network of autonomous devices. The results of these queries have to propagate back to the initiator of the query. Such result propagation is based on the same propagation mechanism offered by TOTA.

Evaluation

Evaluating tuple space based middleware with respect to the hardware phenomena we discussed in section 2.3 we obtain the following results.

Volatile Connections Both Lime and TOTA were designed specifically for mobile (ad-hoc) networks and have considered broken connections in their solution. In Lime the global virtual data space adapts based on the individual ITSes that are available in the communication range of the devices. As described above an ITS is atomically joined and disjoined with the other ITSes in its communication range. Such guaranteed atomic operations generally require a two phase commit protocol. As discussed in section 2.4.4 two phase commit protocols do not scale in the context of volatile connections.

In TOTA tuples are stored each time they enter a tuple space. It is necessary to clean up the tuple spaces after the result of a query has been received. However, no such mechanisms are described to remove tuples. Such mechanisms would require communication and it is not clear how tuples can be removed if a tuple space suddenly becomes disconnected.

Ambient Resources Both Lime and TOTA are able to discover tuple spaces in their ambient and provide event-based mechanisms to take advantage of ambient resources.

Natural Concurrency Tuple space based middleware solutions are based on the coordination primitives offered by Linda [Gel85]. Most of these solutions extend these primitives with asynchronous variants and allow asynchronous events to be subscribed to the tuple space.

Autonomy Although Linda was originally designed from a centralized tuple space perspective, many of the extensions manage to distribute the tuple space. However, in the case of Lime this distribution is associated with the introduction of atomic join and disjoin operations which can create dependencies that harm the autonomy of a device. Nevertheless, other approaches such as in the TOTA middleware do not suffer from this problem. Also, the introduction of asynchronous primitives to interact with tuple space prevents that dependencies between devices are created.

2.7.4 Data Sharing-Oriented Middleware

Data sharing-oriented middleware tries to maximize the autonomy of mobile devices. Thus far we have seen two approaches that are based on shared data structures as a means for communication between devices. First, in Lime a shared tuple space composed of each agent its individual ITS is maintained. This shared data structure is then used as a form of communication between the agents. However, keeping such a data structure consistent requires atomic

operations which do not scale due to the volatile connections characteristic. Second, Rover introduced the concept of a relocatable dynamic object (RDO) as a shared data structure. A RDO can be copied from a server onto a mobile device and the mobile device can autonomously interact with the copy. When the RDO is copied back to the server then conflicts can arise as a result of changes made by other devices that manipulated their copy of the RDO and copied it back to server afterwards. Hence, RDOs allow a form of replicated objects.

Other approaches, such as Coda [SKK+90], Bayou, [TPST98], Rover [JTK97] and XMiddle [ZCME02] are based on a similar replica mechanism. Sharing of resources is achieved by introducing *weak* replica management facilities in the middleware. Weak replicas do not guarantee that all replicas are atomically synchronized such that no two-phase commit protocol is required. As a consequence, synchronization of weak replicas can also lead to a series of conflicts. These conflicts are application-specific and must be resolved at the application level. Hence, the increased autonomy of devices comes at the cost of more complex applications.

Bayou

Bayou [TPST98] middleware provides replicated weakly consistent SQL-based data storage engines. These data storage engines, which can run on mobile devices, are synchronized with one another based on a special anti-entropy protocol [PST+97a]. This protocol is designed such that databases can synchronize with one another sporadically when connectivity is possible between them. Furthermore, the protocol also supports incremental synchronization of the database. Incremental synchronization ensures that if a connection breaks while a database is being synchronized that the synchronization process can resume when connectivity is reestablished. The advantage is that the synchronization process does not need to be repeated. Moreover, the synchronization can be resumed by another replica.

Weak replication can result in conflicts when two conflicting updates are performed on the same data elements. Bayou provides an API such that application specific conflict resolution strategies can be devised. More particularly updates to the database, which are called *bayou writes*, are more than simple SQL statements that update the database. A bayou write contains application-specific meta-data that specifies how conflicts should be solved. An example of a bayou write for a meeting scheduler is shown below:

```
Bayou_Write(
  update = {insert, Meetings, 12/18/95, 10:00am, 60min, Project Meeting: Kevin},
  dependency_check = {
    query = SELECT key FROM Meetings WHERE day = 12/18/95
            AND start < 11:00am AND end > 10:00am,
    expected_result = EMPTY },
  mergeproc = {
    alternates = {12/18/95, 12:00pm};
    newupdate = {};
    FOREACH a IN alternates {
      # check if there would be a conflict
      IF (NOT EMPTY (
        SELECT key FROM Meetings WHERE day = a.date
          AND start < a.time + 60min AND end > a.time))
        CONTINUE;
      # no conflict, can schedule meeting at that time
      newupdate = {insert, Meetings, a.date, a.time, 60min, Project Meeting: Kevin};
```

```
      BREAK;
    }
    IF (newupdate = {}) # no alternate is acceptable
       newupdate = {insert, ErrorLog, 12/18/95, 10:00am, 60min, Project Meeting: Kevin};
    RETURN newupdate;
  }
)
```

Each Bayou write contains a dependency check rule that is executed when the write is applied to a replicated database. Based on this rule the storage engine decides when a write causes a conflict in the database. The merge procedure contains an algorithm that tries to resolve the conflict in the database. In this case the procedure tries to find an alternate schedule for the meeting. The anti-entropy protocol ensures that, in the long run, all replicas of a database will converge to the same state. This protocol is further discussed in section 8.4.

Another issue with weak replication arises when applications consult a replicated database and read data that is changed afterwards as the result of a conflict. For this reason Bayou's API allows to make a distinction between tentative and committed data. The former is data that can change as a consequence of a conflict, whereas the latter is guaranteed to remain stable. Hence, it is up to the application developer to choose the type of data he wants to read from the database such that he can deal with both types of data explicitly. Note that due to the anti-entropy protocol all tentative writes will eventually be committed.

XMiddle

XMiddle [ZCME02] is middleware that enables weak replication of data that is stored in a tree-based XML representation. A host can define a set of access points to the data such that other devices can link with these points and read the associated data. Such an access point actually addresses a branch in the tree representation of the data and the host. When two hosts are in the communication range of one another then hosts can create links to remote branches that were defined as access points. The concept of linking to a tree is similar to the mounting of a remote disk in a distributed operating systems.

The host that defined the access point is called the owner of the branch and only the owner of a branch can define access points on its tree. On the other hand the host that links to the branch is called a peer. Immediately after a peer creates a link to a branch it is downloaded to the peer its device. Although the peer is not the owner of the branch it is able to freely read and modify the branch. If two hosts, either the owner and a peer or two peers, are in the communication range of one another they can reconcile shared branches. To determine if they share common branches both the owner maintains a list of all peers that created a link to its access point and the peer that created a link remembers the owner from which the branch was downloaded. Based on this information together with the path to shared branches two hosts can determine if they share a common tree and synchronize one another once they can communicate. Synchronization of branches always occurs pair-wise. In other words, in a synchronization process two hosts are involved.

An elaborate version mechanism of branches together with a host-to-host reconciliation protocol is used to synchronize hosts. The reconciliation protocol is composed of an application-independent and an application-specific part. The

former part is responsible for updating the corresponding branches that have been modified while two hosts independently modified them. The modifications that need to be reconciled are selected based on the versioning mechanism. Due to the independent manipulation of trees by hosts it is possible that conflicts occur. The example given by Mascolo et al. [ZCME02] is that of a shared shopping basket to which different members of the family can add their weekly purchases. In the case duplicate items are added to the shopping cart a conflict arises that can only be solved in an application-specific manner. Such conflicts are solved by adding a conflict resolution algorithm, called a *resolutor*, to the trees. Possible resolutors are `add`, `last`, `random`, `first`, `greatest`. An example XML tree of such a shopping cart data structure is shown below:

```
<basket>
  <order>
    <product> Milk</product>
    <quantity>
      <howmuch>2</howmuch>
      <resolutor> add </resolutor>
    </quantity>
  </order>
  <order>
    <product> Apple</product>
    <quantity>
      <howmuch>3</howmuch>
      <resolutor> add </resolutor>
    </quantity>
  </order>
</basket>
```

When two conflicting branches are merged and the two branches have the same resolutor, for example `add`, then that resolutor is chosen and the conflict is resolved by adding the quantity of the two orders. In the case the two branches have a different resolutor then the resolutor with the highest priority is chosen. The priority of resolutors is determined by the order by which they were defined. Although the middleware does provide resolutors for resolving conflicts, it does not specify how clients that potentially read parts of the conflicting branches have to deal with the conflicts. Hence, there is no notion of tentative data.

Evaluation

Evaluating data-sharing oriented middleware with respect to the hardware phenomena we discussed in section 2.3 we obtain the following results.

Volatile Connections The replication mechanisms are designed specifically to achieve a high availability of data in the face of volatile connections. The synchronization protocols of these mechanisms are adapted such that there is no need for a two-phase commit protocol.

Ambient Resources Some of the middleware approaches explicitly consider the automatic initiation of the synchronization protocol when replicas are detected in the ambient. However, to the best of our knowledge this detection is transparent for the application. The focus of these middleware approaches rather lays with the synchronization and reconciliation protocols used in the middleware.

Natural Concurrency Shared data sources can be concurrently updated by each autonomous device in the network. This fact together with the weak synchronization protocols of the replicas results in possible conflicts. Such an approach is comparable to optimistic database transaction protocols where no locks are taken and if an inconsistent state is detected when the transaction is committed then the transaction is rolled back. Such an approach is in contrast with the other concurrency approaches we discussed so far because it does not prevent inconsistent states. In other words, the concurrent nature of the devices is fully exploited such with possible inconsistent states as result. These inconsistent states are then resolved with conflict resolution strategies. Hence, natural concurrency is dealt with using optimistic concurrency mechanisms.

Autonomy The replication strategies supported by the middleware we discussed above are specifically designed to maintain the autonomous nature of the devices. Moreover, the replication of data ensures that a device maintains its autonomy even when it is completely disconnected from other devices. As a result devices can continue their operations even when certain resources have become unavailable for communication. These replication strategies can be regarded as the summum bonum of autonomy for devices. Nevertheless, the increased autonomy for devices comes at the cost of more complex programs because they have to deal with inconsistent states and the tentative data that results from accessing inconsistent states.

2.7.5 Summary

Table 2.4 summarizes the evaluations we have made for the different types of middleware. The RPC based middleware solutions each address important individual issues with respect to dealing with the hardware phenomena. Unfortunately, none of them offers support for dealing with the combined hardware phenomena we have discussed in section 2.3. The other types of middleware solution offer better support, but often do not take all hardware phenomena sufficiently into account. Moreover, these middleware solutions do not match very well with the object-oriented paradigm. In these approaches components do not interact based on the paradigm of method invocation. Instead interactions occur based on an alternative form. In the tuple space based approach interaction of processes occurs through the medium of a tuple space. In publish/subscribe based middleware interactions are based on event subscription and the data-oriented middleware is based on data resources that are not modeled with objects. Hence, even though the object paradigm is considered to provide a good foundation to construct distributed systems [BGL98], as discussed in section 2.5, it does not seem to be applied in the context of these middleware approaches. Hence, the aim is to reconcile the better support for the hardware phenomena found in the middleware approaches with the object-oriented programming paradigm.

2.8 Conclusion

In this chapter we have discussed two different types of mobile distributed systems and distilled four phenomena that are exhibited by the hardware compo-

Type	Volatile Connections	Ambient Resources	Autonomy	Natural Concurrency
RPC	Some	Jini	$\emptyset$	$\surd$
Pub./Sub.	?	$\surd$	$\surd$	$\surd$
Tuplespace	Some	$\surd$	$\surd$	$\surd$
Data-Oriented	$\surd$	$\emptyset$	$\surd$	Conflict Resolution

Table 2.4: Summary: Evaluation of Middleware

nents used to compose mobile distributed systems. Next, we have discussed some software issues that arise when developing distributed systems and considered how the object paradigm can help to structure and develop concurrent and distributed software.

There have been a number of proposals for distributed languages that explicitly support open networks. Nevertheless, the current state of the art in distributed languages does not address all the important characteristics that are encountered when developing a nomadic or ad hoc mobile distributed system. On the other hand, middleware approaches offer better, although often incomplete, support to deal with these inherent hardware phenomena of mobile distributed systems. Unfortunately, these approaches do not match well with the object oriented paradigm.

Observations like this justify the need for a new *Ambient-Oriented Programming* paradigm (AmOP for short) that consists of programming languages that explicitly incorporate support for dealing with the observed hardware phenomena in the very heart of their basic computational steps. This is the topic of the following chapter.

Chapter 3

Ambient-Oriented Programming

3.1 Introduction

In the same way that referential transparency can be regarded as a defining property for pure functional programming, this section presents a collection of language design characteristics that define the boundaries of the ambient-oriented programming (AmOP) paradigm [DVM+05]. These characteristics are directly derived from the hardware phenomena we discussed in section 2.3.

The object-oriented paradigm provides good foundations to deal with distribution and its induced concurrency because it successfully aligns encapsulated objects with concurrently running distributed software entities [BGL98]. Therefore, our most basic research assumption is that ambient-oriented programming languages necessarily are concurrent distributed object-oriented programming languages. However, ambient-oriented programming languages differ from conventional distributed concurrent object-oriented programming languages in at least one of the following four ways. These four differences are explained in each of the subsequent sections.

3.2 Classless Object System

Any distributed application must at some point copy objects over the network to be useful. To "prove" this point consider a hypothetical application where all objects are passed as remote references over the network. A remote reference to an object is used to send messages over the network. All arguments of the remote method invocation would then be passed as a remote reference such that the value referred to by a remote reference could never be accessed locally. Hence, as a consequence parameter passing in the context of remote messages, at least *some* objects are to be copied back and forth between remote hosts.

Since an object in a class-based programming language cannot exist without its class, this copying of objects implies that classes have to be copied as well. However, a class is – by definition – an entity that is conceptually shared by all its instances. From a conceptual point of view there is only one sin-

gle version of the class on the network, containing the shared class variables and method implementations. Hence, copying classes over the network causes state consistency problems because objects residing on different machines can independently update (due to the inherent autonomous behavior of devices) a class variable of "their" copy of the class. Moreover, a device might upgrade to a new version of a class thereby "updating" its methods. These are both classical distributed state consistency problems and solving them requires replication machinery. However, in our hardware context consisting of autonomous devices that are connected in a volatile fashion, solving this problem poses some fundamental paradigmatic problems.

By definition, classes impose a sharing relation upon all their instances. This relation is established at object creation time and remains *implicit* throughout the lifetime of *all* its instances. However, because of independent class updates performed by autonomous disconnected devices, two instances of the same class can unexpectedly exhibit different behaviour. In other words, the implicit relation suddenly becomes explicitly detectable. Existing class-based languages do not offer programmers the means to deal with this phenomenon since classes are usually not *fully* reified in the language. For instance, the instance-of link between classes and objects is usually not made explicit in the language precluding transmitted objects from changing their class to a more suitable version upon arrival. Worse, upon detecting inconsistent versions of the same class, no application-independent rule exists to prefer one class over the other. This is witnessed by the data-oriented sharing mechanisms we considered in section 2.7.4. These sharing mechanisms all need application-specific methods to deal with conflicts that arose from independent updates. Hence, in class-based languages the classes are a shared resource that induces conflicts when they are synchronized. To allow programmers to specify how such conflicts are to be resolved, the only viable solution is to *fully* reify classes and the instance-of relation. However, this is easier said than done. Even in the absence of wireless distribution, languages like Smalltalk and CLOS already illustrate that a serious reification of classes and their relation to objects results in extremely complex meta machinery. Hence, from a technical point of view class-based languages induce an extra level of complexity that is placed on the shoulder of the programmer.

Conceptually, a class is a resource, shared across the network by all its instances, that is potentially updated at run-time. However, as discussed above due to volatile connections run-time updates cannot be applied to all devices at the same time. As a consequence of the resulting independent updates the sharing relationship between the class and its instances is (partially) broken. Hence, from a conceptual point of view the concept of a class is broken.

A much simpler solution consists of getting rid of classes and the sharing relation they impose on objects altogether. This is the paradigm defined by prototype-based languages like Self [US87]. In these languages objects are *conceptually* entirely idiosyncratic such that the above problems do not arise. Sharing relations between different prototypes can still be established (such as e.g. traits [UCCH91]) but the point is that these have to be explicitly encoded by the programmer at all times. Surely, a runtime environment can optimise things by sharing properties between different objects. However such a sharing is not part of the language definition and can never be detected by objects. Hence, in a classless object model it never occurs that implicit sharing relationships be-

tween objects become perceptible at the level of the application without having the mechanisms to deal with them explicitly. For these reasons, we have decided to select classless object models for ambient-oriented programming. Note that this confirms the design of the object models found in many existing distributed programming languages such as Emerald [JLHB88], Obliq [Car95] and dSelf [TK02], which all feature classless object models.

3.3 Non-Blocking Communication

The fact that every hardware device is an autonomous computational entity (inducing natural concurrency) combined with the fact that connections are volatile, implies the necessity for non-blocking communication primitives. While evaluating distributed languages and middleware in sections 2.6 and 2.7 we have found that blocking communication primitives can create dependencies between devices, thereby harming the autonomous nature of the devices. This is illustrated by figure 3.1. The figure shows four remote objects, `o1-o4` communicating with one another based on synchronous remote method invocations. To prevent inconsistent states that could result from the concurrency locks are taken and released with each time the objects process a message. This is the behavior exhibited by Java-like concurrency mechanisms when a method is annotated with `synchronized`. With each remote method invocation a dependency is created. For example, `o3` depends on the result of `o4` at the moment it sends `msg3`. The number of dependencies increases with the size of the remote call stack. `o1` ends up depending on `o2`, `o3` and `o4` because the dependencies are transitive. If a the connection between `o3` and `o4` breaks after `o4` received `msg3` but before it returned a result then all objects that are part of that call trace are blocked. What is worse, these objects were locked to avoid race conditions such that these objects are unavailable. These objects can only become available again when the connection between `o3` and `o4` has been restored – or – when a "communication exception" is thrown. However, by throwing a communication exception the code would become cluttered with exception handling code. The other option is that the object waits until the connection has been restored. However, the time needed to restore a connection is often unbounded because the ability to restore the connection can depend on the location of the mobile devices on which the objects reside. In a ubiquitous computing scenario the user is not always aware of the current communication of the devices and as a result he will not necessarily return to his previous location.

A consequence of the dependencies created due to blocking communication is that it is a potential source for (distributed) deadlocks [VA98]. Deadlocks and distributed deadlocks in local networks are not considered to be that harmful, since the cause of the deadlock can relatively easily be debugged with contemporary remote debugging environments. However, in mobile networks, not all parties are necessarily available for communication making the resolution of deadlocks as hard as resolving race conditions.

The root cause of the problem is that the use of locks, which must be used to avoid race conditions on the internal state of an object, interferes with the remote communication mechanisms. An important consideration when designing a concurrency model for a language that is to run on mobile networks, is that the communication mechanism should minimize the duration resources are

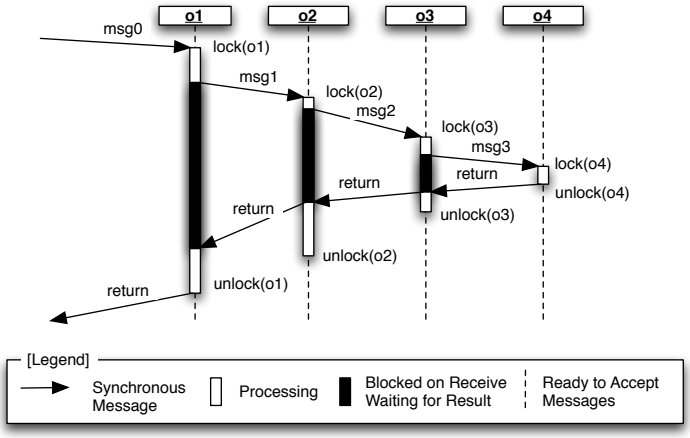

Figure 3.1: Dependencies Created due to Blocking Communication

locked. This is very important, because the extremely high latency of communication (over volatile connections) in mobile networks would diminish the availability of resources. Indeed, having blocking communication primitives would imply a program or device to block upon encountering unstable connections or temporary unavailability of another device. This has previously been remarked on several occasions [MCE02, CNP00, MPR01]. We thus conclude that an ambient-oriented concurrency model is a concurrency model without blocking communication primitives.

Quite often, the issue of non-blocking communication is confused with asynchronous message sending. Asynchronous message sending implies that the send operation is non-blocking, but tells us nothing about the (possibly implicit) receive operation. A typical example of asynchronous send operations combined with blocking "receive" operations is found in the tuple-space based middleware (discussed in section 2.7.3), which provide explicit, blocking receive operations on the tuple-space. Other examples are the `next-value` and `claim` operations to access futures found in ABCL/1 [YBS86] and Argus [Lis92], which were both discussed in section 2.6. Figure 3.2 shows the differences between these communication mechanisms. The figure illustrates that asynchronous systems with a blocking receive operation become prone to the dependency problem as soon as they invoke such a blocking receive operation.

3.4 Reified Communication Traces

Non-blocking communication (both send and receive) combined with the autonomy of the communicating devices implies that they will have to foresee *some* form of handshaking given the fact that these devices are performing a meaningful task together. Since the communication is non-blocking, both senders and receivers will continue their execution irrespective of what happened after a message send. This means that the parties might end up in a state that is no longer consistent with the semantics of whatever the task it is that they

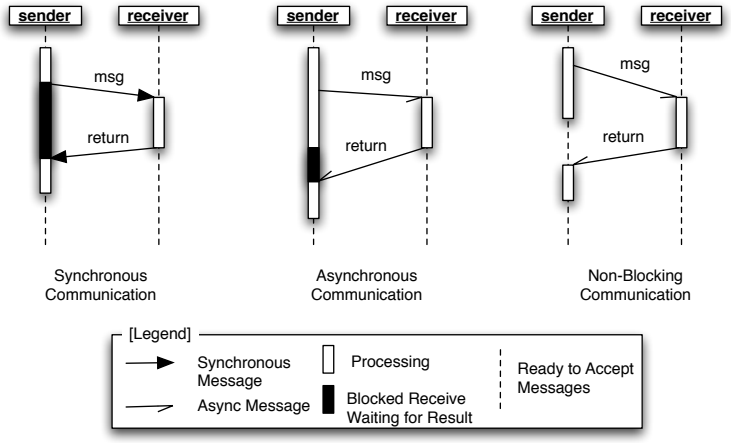

Figure 3.2: Synchronous Communication vs. Asynchronous Communication vs. Non-Blocking Communication

are solving. Whenever such an inconsistency is detected, the parties must be able to restore their state to whatever previous consistent state they were in, such that they can decide what to do based on that final consistent state they agreed upon. Examples of the latter could be overruling one of the two computations or deciding together on a new state with which both parties can resume their computation. Therefore, a programming language in the ambient-oriented paradigm has to provide us with reversibility provisions giving programmers a way to manipulate their execution state based on an *explicit representation* (i.e. a reification) of the communication details that led to the inconsistent state. This explicit representation will allow them to take the appropriate actions to reverse (part of) the computation. Notice that any implicit way to prevent the communicating parties from ending up in an inconsistent state implies that communication primitives are blocking, which was precluded above. Having an explicitly reified representation of whatever communication that happened, allows a device to properly recover from an inconsistency by reversing part of its computation.

A second argument in favor of reified communication traces is the following. Several degrees of message delivery guarantees can be associated with non-blocking communication. For example, in the many-to-many invocations library [KB02], where all communication occurs via asynchronous messages, there are no delivery guarantees. When a message is sent and there is no process listening for messages, the message is lost. Such communication paradigm is lightweight with respect to the usage of resources and is suitable when no delivery guarantees are to be met. On the other end of the spectrum there is the actor model, where all asynchronous messages that are sent must eventually be received [Agh86]. Such an approach is perhaps feasible when there are abundant resources, but in the context of mobile computing, where devices have scarce resources, it is clear that such an approach is not practicable. This shows that there is no single "right" message delivery guarantee policy because a tradeoff will have to be made based on the requirements of the application and on the available

resources. Programming languages belonging to the ambient-oriented paradigm should make this tradeoff possible instead of imposing a single strategy. Explicit control over the communication traces allows one to make the tradeoff between different delivery guarantees.

3.5 Reified Environmental Context

The fact that hardware devices are autonomous, combined with the fact that resources are dynamically detected as the devices are roaming means that all devices potentially have the same capabilities to interact with each other directly without relying on a third party. This is in contrast to client-server communication models where clients usually interact through the mediation of a server (such as is the case with chat servers or white boards). The fact that communicating parties do not need an explicit reference to each other (whether directly or indirectly through a server) requires what is known as *distributed naming* [Gel85]. For example, in tuple-space based middleware this property is achieved, because a process can publish data in a tuple space, which can then be consulted by the other processes based on a pattern-matching basis. Another example is many-to-many invocations [KB02], where broadcasts to all objects implementing a certain interface can be expressed. Distributed naming is especially important in the context of ad hoc distributed systems, because it provides a mechanism to communicate without knowing the address of an ambient resource. Ambient resources can be perceived by AmOP applications if the environmental context of the device is reified. A *reified environmental context* allows AmOP applications to detect when ambient resources appear and disappear in the environment and can also entail the reification of other useful environmental context such as the signal strength of a wireless connection. It is important that this reification preserves the autonomous nature of the devices. Hence, AmOP programs should be able to self-sufficiently reify the environmental context.

We are not arguing that all ambient-oriented applications have to rely on a reified environmental context. It is perfectly possible that a programmer (or even a suite of running processes) sets up a server for the purposes of a certain application. However, an ambient-oriented programming language should allow applications to rely on reified environmental context should this be required. In other words, the acquaintances of an object must be dynamically manageable. We will also refer to this property as *ambient acquaintance management*.

3.6 Software Platforms Revisited

Based on the AmOP criteria we distilled above we can revisit the evaluation of the software platforms we evaluated in the previous chapter.

3.6.1 Distributed Languages

In section 2.6 we evaluated distributed languages with respect to the hardware phenomena from section 2.3. We concluded that distributed languages in general do not support ambient resources and volatile connections. However, the languages designed for open networks had a concurrency model that preserved

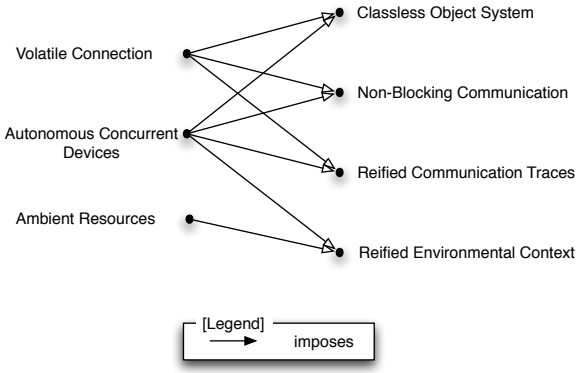

Figure 3.3: Hardware Phenomena inducing AmOP Characteristics

Language	Classless Object Model	Non-Blocking Communication	Reified Comm. Traces	Reified Environment
ABCL	√	∅	∅	∅
Argus	∅	∅	∅	∅
E	√	√	∅	∅
Salsa	∅	√	∅	∅
nesC	N/A	√	∅	√

Table 3.1: Evaluation of Distributed Languages based on AmOP Criteria

the autonomy of the devices. Below, we revisit this evaluation of distributed languages in the context of the AmOP criteria. The results are summarized in table 3.1.

Classless Object Model

Of the languages we evaluated, ABCL and E are based on a classless object model. Argus is an extension of the CLU language which features a class-based object model. The Salsa language is implemented as a pre-compiler for Java and inherits Java's object model.

Non-Blocking Communication

Both ABCL and Argus feature asynchronous method invocations. These invocations result in futures and promises, respectively, that are proxies for the return-values. These proxies can be queried for the result they represent. In the case of ABCL such a query is done based on the `next-value` operation and in Argus a `claim` operation is introduced. Both these operations block if the future or promise has not been resolved. Hence, these operations introduce a blocking "receive" primitive in the language.

E also features asynchronous method invocations that result in promises. However, instead of introducing a `claim` operation to get the result that the promise represents a `when` operation has been introduced. This `when` operation schedules a closure in the queue that is invoked when the promise is resolved. Hence, E fulfills the non-blocking communication criterion.

Type	Classless Object Model	Non-Blocking Communication	Reified Comm. Traces	Reified Environment
RPC	∅	∅	Some	Some
Pub./Sub.	∅	√	Some Partial	√
Tuplespace	∅	√	Partial	√
Data-Oriented	∅	√	Some Partial	∅

Table 3.2: Evaluation of Middleware based on AmOP Criteria

Salsa introduced linguistic abstractions for dealing with the results of asynchronous method invocations based on different continuation passing style. These continuation passing style messages prevent the introduction of blocking "receive" operations such that Salsa fulfills the non-blocking communication criterion.

Reified Communication Traces

To the best of our knowledge no distributed language reifies the object's communication trace.

Reified Environmental Context

In the distributed languages we have reviewed only nesC can provide a reification of the environmental context using the broadcast component.

3.6.2 Middleware

In section 2.7 we evaluated middleware for mobile computing with respect to the hardware phenomena from section 2.3. We concluded that the middleware approaches better support the hardware phenomena than distributed languages, but do not reconcile well with the object-oriented paradigm. Below, we revisit the evaluation of middleware in the context of the AmOP criteria. The results are summarized in table 3.2.

Classless Object Model

None of the middleware approaches we discussed are based on classless object models.

Non-Blocking Communication

The RPC-based approaches are mostly based on asynchronous communication but with a blocking "receive" operation. The publish/subscribe paradigm features interactions based on subscribed event notifications. This interaction style is an inherent form of asynchronous communication. Publish/subscribe based middleware usually do not introduce a blocking "receive" operation. In tuplespace based middleware communication is expressed in the form of reads and writes to the tuple space. Linda [Gel85], on which all tuple space middleware is based, features a blocking "read" operation. However, the tuple space middleware designed for mobile computing often introduce non-blocking "read" operations in the form of events that are subscribed on the tuple space. The

data-oriented middleware approaches are based on replicated data structures. Hence, these data structures reside on the same device as the application such that interactions can occur through local communication and do not rely on blocking remote communication.

Reified Communication Traces

In tuple space based approaches, discussed in section 2.7.3, the tuple space can be regarded as a partially reified communication trace. For example, in Lime [MPR01] the tuple space contains tuples that are moved upon connection from one tuple space to another. An agent can thus consult its own tuple space and make changes after the communication occurred. In most other tuple space based approaches similar actions can be undertaken. Hence, through the access to the tuple space it is possible to change the delivery semantics between multiple tuple spaces. In the RPC-based approaches, Rover introduces logging of remote communication [JdLT+95] to support customized strategies for the reconciliation of RDO objects copied onto a client. In Bayou [TPST98], the notion of explicit write-logs is introduced to enable synchronization of different replicas.

Reified Environmental Context

The RPC-based middleware is mostly designed for nomadic distributed systems. The only exception is Jini which has been designed for ad-hoc distributed systems and reifies the environmental context. In STEAM [MC02], which is a publish/subscribe middleware, introduces the notion of proximity groups to enable communication based on the geographical location of devices. These proximity groups can be regarded as a reified environmental context. Both Lime [MPR01] and TOTA [MZ04] expose the environmental context to the applications. In Lime this is done through a dedicated tuple-space, called `LimeSystem`, that is maintained by the system. The system continuously updates this tuple space with the other tuple spaces that are in the communication range. In TOTA changes in the ambient can be monitored by subscribing to an event. In the data-sharing oriented middleware the environment is not explicitly exposed to the application. However, in XMiddle [ZCME02] the synchronization and reconciliation process of data trees starts automatically when a device with shared data appears in the communication range of another device. Nevertheless, to the best of our knowledge this information is not made explicitly available to the application.

3.7 Discussion

An important consequence of each of these AmOP characteristics is that they influence the structure of the software. For example, the requirement of non-blocking communication primitives leads to the use of event-handlers. It is widely acknowledged that event-handlers induce complex program structures because they obfuscate the control flow of an application. The reason is that event-driven approaches need to emulate an object-oriented programming style manually. I.e. “calling” a method on an object has to be done by emitting an event. The “return” value of this method invocation has to be returned

by the object by emitting another event with the result. This emulation of an object-oriented programming style forces the developer to manually encode a continuation passing style which results in a complex program structure. For this reason it is necessary to come up with language features that allow us to reconcile the AmOP paradigm with an object-oriented programming methodology.

The distributed languages based on the actor model, Salsa and E, show that language constructs can be used to reduce complexity that emerges as a consequence of non-blocking communication primitives. This is further discussed in chapter 7.

3.8 Conclusion

In section 2.3 we have considered important phenomena based on the hardware components used to construct mobile distributed systems. After having analyzed the implications of these hardware phenomena on the design of programming languages and middleware in sections 2.6 and 2.7, we have distilled the above four characteristics. We will henceforth refer to programming languages that adhere to them as *Ambient-oriented Programming Languages*. Surely, it is impossible to prove that these are strictly necessary characteristics for writing the applications we target. After all, AmOP does not transcend Turing equivalence. However, we do claim that an AmOP language will greatly enhance the construction of such applications because their distribution characteristics are designed with respect to the hardware phenomena presented in section 2.3. AmOP languages incorporate transient disconnections and evolving acquaintance relationships in the heart of their computational model.

The connections between the hardware phenomena and the characteristics are illustrated by figure 3.3. The figure shows that there is a strong coupling between volatile connections and autonomous concurrent devices on the one hand and the first three AmOP characteristics on the other hand. Dealing with ambient resources in this hardware context leads to the need for a reified environmental context.

Based on tables 3.1 and 3.2 we conclude that the state of the art in distributed languages does not conform to all characteristics of AmOP. On the one hand, languages such as Argus and ABCL do not support the non-blocking communication characteristic. On the other hand, languages for open networks based on the actor model usually have the non-blocking communication characteristic, but do not allow for a reified environmental context and are not equipped with reversibility provisions. The nesC language does have some support for reified environmental context but does not allow for high-level non-blocking communication between components. In the middleware approaches we have found a number of approaches, especially those based on tuple spaces, that come close to support the hardware phenomena, but unfortunately these approaches do not match well with the object paradigm.

At this point in the dissertation it is useful to take a step back and state the plan for the remaining chapters of this dissertation based on the conclusions in this chapter:

- In the next chapter we will consider an extension of the actor model in the context of AmOP. We have chosen for the actor model because it

is the concurrency and distribution model that influenced both E and Salsa, which both supported the non-blocking communication criterion. Moreover, the model is specifically conceived in the context of the object-oriented programming paradigm. However, the actor model needs to be extended to support the remaining AmOP criteria. This exercise together with the results from this chapter address the third research goal we set in section 1.1.3.

- In chapter 5 we informally discuss the semantics of a language, called AmbientTalk, which is based on this extended actor model and therefore supports the AmOP criteria. AmbientTalk allows us to write our first AmOP applications such that we can gain insight in the structure of AmOP applications. This chapter addresses the second research goal we set in section 1.1.3. Based on these insights we can identify a number of limitations with respect to the structure of AmOP applications.

- In chapter 6 we address two issues with respect to AmbientTalk. First, thus far we have only defined AmbientTalk informally. We define the semantics of AmbientTalk through the implementation of a metacircular version of AmbientTalk. Based on this version of AmbientTalk we are also able to define reflective hooks in the language such that we can address the limitations we found in the previous chapter.

- In chapter 7 we address the first research goal of this dissertation (discussed in section 1.1.3). We use the reflective hooks, defined in the previous chapter, to extend AmbientTalk with language features such that AmOP applications can be better structured. These language features will focus on addressing the hardware phenomena we discussed in section 2.3. What is more, these language features are all based on the AmOP criteria we distilled in this chapter and therefore serve as a validation. We then create an AmOP application and structure it based on some of the language features we introduced. We further discuss a similar application written in Java and make a comparative analysis between both applications.

- In chapter 8 we further show the expressive power of AmbientTalk by discussing the implementation of a number of advanced language constructs. These language constructs are based on the AmOP criteria and allow us to address some specific issues encountered while developing advanced AmOP applications.

Chapter 4

The Ambient Actor Model

This chapter presents the ambient actor model, a basic computational model for concurrency and distribution, that adheres to the AmOP criteria that we have argued for in the previous chapter. This model is embedded in the programming kernel we present in the next chapter that is then further used in the remainder of this dissertation to show that the criteria underlie the language constructs that facilitate the development of ambient-oriented software.

4.1 Introduction

The goal of this chapter is to develop a suitable model for concurrency and distribution for AmOP applications. While evaluating the state of the art in chapter 2 we have encountered three well developed models. In the camp of the distributed languages (discussed in section 2.6) we have encountered the ABCL and the actor model and in the camp of the middleware we have encountered the tuple spaces paradigm (discussed in section 2.7.3). Other approaches were based on a thread based model for concurrency and communication primitives to support distribution. Of these models the ABCL and actor model best supported the object-oriented paradigm. The reason we make an explicit distinction between the ABCL and actor model is that, although the name suggests otherwise, the ABCL language is in fact an extension of the actor model. Indeed, the authors [YBS86] note that although the roots of ABCL lie in the actor model the resulting concurrency model differs from the actor model. This difference, which was notable in the conclusion of the evaluation of the distributed languages, is that the languages E and Salsa naturally supported the autonomous nature of the hardware as opposed to the ABCL language family. Hence, the languages based on the pure actor model did not suffer from this limitation.

For this reason we further assess the actor model that was developed by Hewitt [Hew77] in the late seventies and later further developed by Agha [Agh86, AH88]. It was only in the late nineties that Agha et. al published a description of the operational semantics [AMST97] of an actor system. In this chapter we will rely on this operational semantics, because it gives a clear specification about what lies at the heart of the actor model.

In the next section we will give an overview of the actor paradigm as an extension of a functional model based on the λ-calculus. The actor model ex-

tends such a functional model and introduces a number of basic operators that upgrade it to an object-oriented model for parallel and distributed computation.

The actor model is designed for open distributed networks, such as the internet, where communication partners are sometimes unavailable for a short period of time. However, in a mobile network communication can be interrupted for a longer period or even indefinitely. In section 4.3 we evaluate the actor model in the context of mobile distributed systems by analyzing it against the AmOP criteria that we argued for in the previous chapter. Based on these criteria we will come to conclude that the actor model already partially adheres to some of these criteria. This conclusions also indicates that the actor model can be extended such that it fully conforms to the AmOP criteria.

Nevertheless, in section 4.4 we first consider a proper extension of the actor model, called actorspaces, that supports a form of distributed naming. As discussed in section 3.5 distributed naming is a useful abstraction to address actors based on a specification rather than an explicit reference to an actor. However, we will come to conclude that the distributed naming scheme of the actorspace model does not provide the necessary flexibility and autonomy needed in the context of mobile distributed systems.

After this observation we turn back to the actor model in section 4.5 and extend the operational semantics of the actor model. Afterwards we discuss how the extensions translate back to the context of the AmOP criteria. In section 4.6 we show the use of the extensions in the actor model and develop a first AmOP application based on the extensions.

4.2 Actors

The Actor programming language [Agh90] was designed for use in open distributed network environments (i.e. the internet). A distributed application is modelled with actors that are distributed throughout the network. Communication between actors occurs solely with asynchronous message passing. Figure 4.1 shows the conceptual representation of an actor. Each actor has a behavior associated with it. The behavior defines how an actor handles incoming messages. Incoming messages are handled by the actor its own thread of control. An actor is fully encapsulated and can only be addressed by other actors through its mailbox. In other words if an actor sends a message to another actor or itself it always places a message in this mailbox. A message is transparently and non-deterministically selected from the mailbox and processed according to the actor its behavior. Fairness is assumed such that all messages are eventually processed.

An actor-based programming environment consists of two elements: 1) an actor programming language that provides the necessary constructs for expressing the concurrent and distributed computation - and 2) an actor system that provides the run-time environment of actors and communication infrastructure. Both elements are discussed below.

4.2.1 The Actor Programming Language

The semantics of the actor programming language [AMST97] is defined as an extension of the operational semantics of a simple functional language. In the

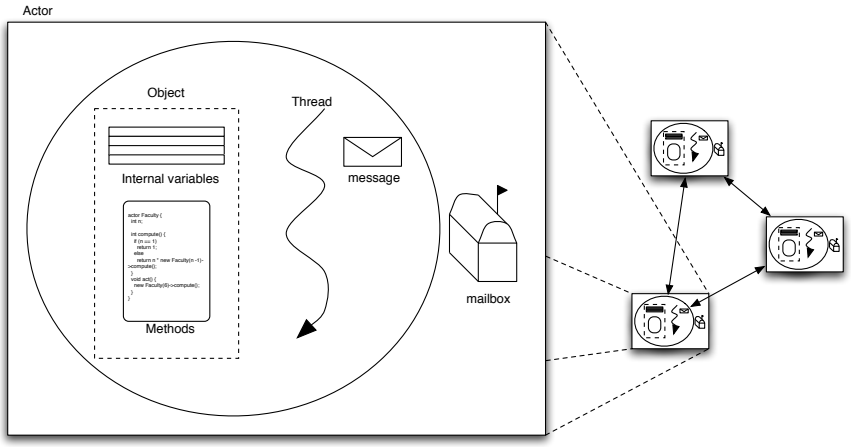

Figure 4.1: Conceptual Representation of Actors

semantics functions are used to model the behavior of an actor. Such a function takes one parameter, a message, and based on this parameter a number of expressions are evaluated. In the operational semantics of the actor language the functional language is conceptualized by the untyped lambda calculus [Mog89]. That functional language is extended with three actor primitives to support programming in a distributed environment:

- New actors can be created using the `letactor` primitive. The `letactor` primitive takes one argument, a function that is the initial behavior of that actor and returns the mail address of an actor.
- Messages are sent to known actors using the `send` primitive. The `send` primitive takes two arguments, the recipient's actor address and a message. Such a message can contain the address of other actors.
- An actor can modify its own behavior using the `become` primitive. The `become` primitive takes one argument, a function that is the new behavior of the actor. There is no shared data between actors.

These primitives are illustrated by the example shown below, which is an implementation of an ML reference cell expressed in the actor language defined by the operational semantics. This cell example also illustrates how the become operator can be used to model state.

B_{cell} = rec(λb.λc.λm.
    if(get?(m),
       seq(become(b(c)), send(cust(m), c)),
       if(set?(m),
          become(b(contents(m))),
          become(b(c)))))

In the λ-calculus functions are first-class entities and do not necessarily have a name. Such anonymous functions are of the form $\lambda a.exp$ where `a` is an argument and `exp` is the body of the function. The last expression that is evaluated

determines the return-value of the function. In the λ-calculus functions take only one argument at a time. Multiple arguments are simulated by currying. This is the process where one function returns another function that takes one argument and this process is repeated for all arguments.

The function B_{cell} describes the behavior of a cell actor. The function call to `rec` calculates the fixed-point of a function [MT91] and calls the λ-expression with that fixed-point as its argument. Hence, B_{cell} refers to a function which takes two arguments `c` and `m` as its arguments and where `b` has been bound to the fixed point in the lexical environment of that function. The argument `c` refers to the contents of the cell and the variable `m` refers to the message that an actor receives.

The cell-behavior responds to two kinds of messages, `get`- and `set`-messages. The `get`-messages are responded to in two steps: first, the `become` operation is used to define the replacement behavior of the actor. The replacement behavior is defined with a recursive call through the fixed-point and the contents of the cell remains unchanged. Hence, the replacement behavior of the actor will refer to a function that takes one argument `m` that has `b` bound to the fix-point and `c` bound to the original contents of the cell. Next, the contents of the cell is sent to the customer of the message. The `set`-messages are responded to by changing the function to a new one with the variable `c` bound to the contents found in the message `m`.

The `become` is placed before `send` to handle `get`-messages is because this order of the statements influences the degree of concurrency. Once the `become` operation has been performed the actor can process its next message while the expressions that follow the `become` are executed concurrently. The `become` operation dynamically creates a new anonymous[1] actor which executes the following expressions, in this case the `send`, while the current actor processes its next message in the context of its new behavior. Accordingly the actor model supports intra-object concurrency. What is more, this intra-object concurrency does not lead to race conditions on the internal state of an actor. The main reason for this is that the state change (achieved by installing another function using the `become` operation) of an actor occurs in a single operation, because naturally the functional language does not include assignments. Hence, only a `become` can change the state of an actor.

The function which describes the behaviour of the cell can now be used to initialize the actor:

$$letactor(a := B_{cell}(0))_e \texttt{ where}$$
$$e = \texttt{seq(send(a, mkset(3)), send(a, mkset(4)), send(a, mkget(a)))}$$

The `letactor` primitive binds a new cell actor to the variable `a` and the expression `e` is evaluated in this context. The `set`-messages are created using `mkset` and similarly `get`-messages are created using `mkget`. The actor model does not define the order in which messages are consumed and accordingly, the response to the `get`-message will depending on the order of the messages be 0, 3 or 4.

[1] An anonymous actor is an actor whose address is not known by any other actor in the system.

4.2.2 Actor Systems

The actor language relies on an actor system to support the parallelism and communication between actors. An actor system hosts multiple actor objects. These actor object may concurrently process messages from one another, irrespective of their individual states. Conceptually, an *actor system* can be modelled as a message set and the behavior of the actors running on the system. This message set contains two types of messages: 1) messages received, but not yet processed and 2) messages sent, but not yet transmitted. Both types are discussed below:

- A message, whose target address is that of an actor running in the actor system, is taken from the message set and passed as an argument to the function that is assigned to the target actor address. When the message set is empty the system waits for a new message to arrive. As explained above, an actor can handle the next message from the moment it has performed the `become` operation.
- When a message is sent then the message is put in the message set. When the target actor address of the message is that of an actor running on another actor system then the message is transparently transferred to the message set of that actor system. The operational semantics of the actor language do not define any order in which the messages from the message set are processed[2], but it assumes fairness so that no starvation can occur.

4.3 Evaluation of Actors for Ambient-Oriented Programming

In this section we evaluate the actor model against the AmOP criteria we have set for ambient-oriented programming languages in the previous chapter. We discuss why the concepts of concurrency and distribution found in the actor model are a good starting point to build an ambient-oriented programming language. Next to that we also pinpoint a number of limitations in the context of mobile distributed systems.

4.3.1 Evaluation #1: The Object Model

In the example above we have seen that state in the actor model was modeled as a λ-function. These λ-function are better known as closures. Both terms are interchangeably used in the rest of this chapter. A closure consists of a pair of pointers, one pointer refers to the code of the function and another pointer refers to the lexical environment of the function. Nested closures can be used to model and create objects, because their lexical environments can serve as a representation to encapsulate state. A closure will have a pointer to a lexical environment extended with the formal parameters bound to the actual parameters. In the cell example above the contents of the cell is encapsulated in the lexical environment of the closure that takes `m` as an argument. The behavior of an actor can be represented by a closure that takes a message as its argument and acts as a dispatch function. The state of the actor is then encapsulated in the lexical environment pointed to by that closure.

[2]Hence, the name message *set* as opposed to message *queue*

Despite the ability to model objects naturally with closures, one could still argue that the actor model lacks a true object model because one still has to manually encode one's own dispatch function. Nevertheless, this is a problem that can be easily alleviated by means of syntactic sugar as illustrated by the language E [Mil04], which models its objects with closures. What is more, the resulting object model relates more to an object-based model than a class-based model [Dic92]. The reason that closures form a good basis for a class-less object model is that they hold both the object state and behavior. Classes, on the other hand, only hold the behavior of the object. In the actor model objects are created by invoking a closure that returns a closure that takes a message as its argument. The former closure can be regarded as an object generator function which creates objects.

Remember from section 3.2 that a classless object model was argued for as a required property of ambient-oriented programming languages, because it induces less sharing problems when objects are copied back and forth over the network. In the actor model a more conservative approach is chosen: the actor model explicitly states that no expressions containing closures can be communicated. In other words, since objects are represented as closures with a dispatch-function we cannot copy objects over a network. Agha et al. [AMST97] argue that this is not a serious limitation, because the actor address can be communicated instead and by sending messages to this actor address we can access the lambda-abstraction. Nevertheless, an actor address is merely a reference to an actor rather than a copy of the actor. Hence, if the actor referred to by the actor address is temporarily unavailable for communication we do not have access to the state and behavior referred to by the closure. However, contemporary languages based on the actor model, such as Salsa [VA01] and E [MTS05], have shown that this limitation can be omitted in practice.

4.3.2 Evaluation #2: Non-Blocking Actor Communication

In the actor model communication amongst actors only occurs by means of the `send` primitive. Hence, no implicit communication occurs such as through shared data. This send operation is non-blocking and this is a requirement of the ambient oriented programming paradigm to preserve the autonomy of a mobile device as explained in the previous chapter. Next to the non-blocking `send` primitive there is also a requirement for a non-blocking receive primitive. In the actor model there is no receive primitive available in the language. That is, a program cannot be written such that the control flow of the actor is blocked until a specific message is received from another actor. Thus, once an actor starts processing a message it will finish executing that message without waiting for the availability of other actors that are communicated with.

4.3.3 Evaluation #3: Reified Communication Traces

The actor model was designed for open distributed networks, where communication partners are sometimes unavailable for a short period of time. In the actor model delivery guarantees are achieved through the use of message sets: messages sent to actors running on an actor system that is unavailable are kept

in the message set until the actor system becomes available again for communication. The actor model has useful properties in the context of mobile (ad-hoc) networks, because of these message sets. Each actor system has its own message set that contains the messages sent, but not yet transmitted to another actor system and the messages received, but not yet processed. Message sets have two favorable properties with respect to this type of networks:

- Each actor system is equipped with its own message set. An actor system is therefore self-sufficient and does not rely on a general server infrastructure from the environment for its communication.
- A message set enables transparent asynchronous communication in intermittently connected environments, because a message is transparently and automatically transferred from the message set of the sender to the message set of the receiver whenever communication is possible. Hence, the connection between two actor systems is automatically restored after it was broken. This is important because it permits transparent communication in networks where failures are common rather than the exception, whereas many other asynchronous protocols put the burden of expressing delivery guarantees on the developer.

These message sets can be regarded as a part of the communication traces of such an actor. Unfortunately, in the actor model these communication traces are not first class entities. As a result, one cannot intervene in the communication process of an actor and only a single delivery strategy is employed, namely eventual delivery of messages. In other words, its communication layer is based on the assumption that connections between actor systems will always be restored after they are broken. Such an assumption is attainable in an open distributed network such as the internet, but considering mobile networks it is impossible to attain, because connections often break but are not necessarily restored due to the unpredictable mobility of the devices. This mobility results from the devices their autonomous concurrent nature and the fact that resources coincide with the immediate ambient in which the device is located, as discussed in section 2.3.

Reified communication traces would allow one to build abstractions with different delivery strategies. What is more, as argued in section 3.4, reified communication traces are also needed to provide programming abstractions to deal with possible conflicts that result from volatile connections. The actor model does not provide any constructs or abstractions to deal with these conflicts such that they would need to be solved in an ad-hoc manner.

4.3.4 Evaluation #4: Reified Environmental Context

Another criterion of ambient-oriented programming languages is that the environmental context of mobile devices should be reified so that abstractions can be introduced to deal with the continuously changing ambient resources used by a mobile device. In the actor model all resources that can be shared are modeled as actors and actors are referred to by actor addresses. Actors can only communicate with one another if they have an actor mail address. Miller et al. [Mil04] noted that in capability secure languages, such as the actor model, actors can only make acquaintance with one another in four modes:

- Connectivity by Introduction: a message sent to an actor can contain the address of another actor.
- Connectivity by Parenthood: when an actor creates another actor using the `letactor` primitive, then the creating actor has the address of the new actor.
- Connectivity by Endowment: when a new actor is initialized with the address of another actor.
- Connectivity by Initial Conditions: when an actor system is initialized for the very first time, then there is a bootstrapping phase needed so that the very first actor can be created and such that it has the necessary actor references in order to initialize properly so that the other connectivity rules can be applied.

The rules above identify a limitation: the actor model does not provide a means for an actor to independently get a reference to a remote actor in its direct ambient. In other words, if two devices move in the communication range of one another then there is no rule that permits one actor running on a device to get a reference to another actor running on the other machine, unless both actors would have a reference to a third party that could act as a middleman. Such a middleman could be a naming server or name registry which is used in middleware approaches, such as Java RMI and CORBA. However, such a middleman would imply that infrastructure always needs to be present in the ambient. This would not only conflict with the autonomy of mobile devices as explained in section 2.3 but the actor address of the software running on that infrastructure would need to be bootstrapped in all mobile devices.

4.3.5 Summary

We have evaluated the actor model with respect to the criteria set by ambient-oriented programming paradigm and have uncovered a number of limitations that make it either impossible to use the current actor model in mobile (ad-hoc) networks or difficult to program in the actor language. The limitations are summarized below:

- The actor model does not allow closures (which are used to represent objects) to be communicated over the network. This limitations introduces practical problems, but are overcome in contemporary distributed programming languages based on the actor model.
- The actor model does not define how actor addresses can be resolved. Actors need to know their available communication partners, especially because they change frequently when the user arrives at a new location. We need an abstract way to reference the set of actors in that ambient that are available for communication. Currently, the actors have no means to find each other when mobile devices are in the communication range of one another such that ambient resources cannot be detected.
- The model does not support volatile connections with disconnections over a longer period. The actor model assumes that messages sent will eventu-

ally be received. In a mobile network this precondition cannot be guaranteed anymore. There is a need for more explicit control over the delivery of messages. Moreover, conflicts that arise due to the disconnectivity of actors have to be dealt with in an ad-hoc manner.

Despite these limitations the actor model adheres to the non-blocking communication criterion. What is more, the actor model matches well with the object paradigm, which was one of our basic research assumptions for the AmOP criteria. Moreover, the object model featured in the actor model is based on objects rather than classes. These are unique traits compared to the other models for concurrency and distribution we have encountered while evaluating the state of the art (sections 2.6 and 2.7). Hence, although the actor model does not completely support the AmOP paradigm we believe it provides a solid starting point to extend it such that the missing criteria, namely the reification of communication traces and environmental context, are supported too.

4.4 Evaluation of the ActorSpace Model

An extension of the actor model, named actorspaces, was proposed by Agha and Callsen [AC93] and defines how actors can be resolved in a distributed namespace. As discussed in section 3.5 distributed naming is a useful abstraction to address actors based on a specification rather than an explicit reference to an actor.

Actorspaces introduce a form of distributed naming into the actor paradigm. The actorspace model extends the actor model with three new concepts:

- *Attributes*: In the actorspace model, actors can be addressed by their actor address or by attributes. Attributes are patterns which provide an abstract specification of an actor. In contrast to an actor address, which is associated with exactly one actor, patterns can denote a group of actors; namely all actors which fulfill the abstract specification determined by the attributes.

- *ActorSpaces*: An actorspace is a computationally passive entity that determines the scope in which the pattern matching of attributes occurs. Actors and actorspaces can be made (in)visible in the context of an actorspace. Hence, the entities (actors and actorspaces) that are matched by a pattern, which is parameterized with an actorspace, will be limited to the entities that reside in that actorspace.

- *Capabilities*: The model of security is based on capabilities [Lev84]. A capability is an unforgeable key which can be created dynamically, compared and communicated over the network by actors. The correct capability is needed to change properties with respect to the distributed naming of an actor or actorspace.

These concepts are introduced in the programming language by introducing new primitives and changing existing ones. They are summarized below: `create-space` takes a capability as an argument and creates an actorspace and returns an address for this actorspace. `make-visible` and `make-invisible` are used to add and remove actors and actorspaces to an actorspace. There is also

a primitive `change-attributes` that can change the attributes to actors and actorspaces. The `send` primitive in the actorspace model differs from the standard actor model in that it can not only take an actor address as an argument, but also a pattern expression of the form `pattern@space`. When multiple actors in the actorspace match the pattern, then one actor is non-deterministically chosen. Besides `send`, another communication primitive `broadcast` has been introduced. This primitive takes a pattern as its argument, but the difference is that the message is sent to all actors in an actorspace that match the pattern. No global or partial order is assumed on both message sending primitives.

Some of the primitives we have discussed above involve the use of capabilities. Capabilities can be associated with both actorspaces and actors in the model and actors need to have the correct capability in order to manipulate the naming properties (using the primitives `make-visible`, `make-invisible` and `change-attributes`) of these entities. In the actorspace model a number of actors play the role of a *manager*, which regulate the naming properties of actorspaces.

Although the actorspace model introduces an interesting model to deal with distributed naming problems of the actor model, there are a number of limitations that make the concepts introduced in the actorspace model impossible to use in the context of mobile distributed systems. The problems originate from the fact that mobile distributed systems typically involve network partitions as a result of the volatile connections (discussed in section 2.3). For example, suppose a number of actors are contained in an actorspace but some of these actors run on mobile devices that are currently not in the communication range. Hence, the actorspace is partitioned and it is undefined what happens when messages are sent to the actors in that actorspace. Also, it is not clear on which device the data structure that holds the configuration of the actorspace should be placed. When the node that maintains that data structure is not in the communication range at the moment a `send` or `broadcast` operation is performed, then the semantics is undefined. Another point where semantics of the operations of the actorspace model is not clear is when managers change the configuration of an actorspace in the face of such a network partition. Note that replication of the actorspace's datastructure is not feasible, because an actorspace can be manipulated at runtime and keeping the replicas up-to-date and provide consistent access does not scale and leads to inconsistencies.

The main problem is that the tuple space model was not designed for network partitions, which result from volatile connections. Based on this evaluation we have decided to step back to the standard actor model and extend it such that it adheres to all AmOP criteria.

4.5 The Ambient Actor Model

In this section we introduce the *ambient actor model*, an extension of the operational semantics and the syntax of the standard actor model such that the limitations, pointed out in the previous sections of the actor model and the actorspace model, are resolved. We will refrain from giving all the definitions of the standard actor model, which can be found in [AMST97]. We will however repeat definitions that we adapted to extend the operational semantics of the actor model or when they are essential to understanding the extensions.

The main addition to the actor model is the introduction of *explicit* mailboxes [DV04] for each actor. A number of mailboxes are used within the model to guarantee communication between local and remote actors. These are the `inbox`, which keeps track of incoming messages, the `outbox`, which keeps track of messages that should be delivered, the `sent-` and the `rcvbox` which keep track of, respectively, which messages have been sent and which messages have been processed by an actor. In this section we show that the introduction of these four first-class mailboxes addresses the reification of the communication traces, which was one of the missing AmOP criteria we identified in section 4.3. We will show that through the manipulation of these mailboxes actors are able to adapt their delivery strategy based on the needs of the AmOP application. What is more, actors can use the mailboxes to determine their communication state and as such implement customized schemes based on this state.

Aside from these mailboxes four other mailboxes are introduced to reify the environmental context of actors. This type of reification is the last missing AmOP criterion we identified in section 4.3. Actors are usually interested in specific resources from the ambient and in the (dis)appearance of these resources. For this reason, two mailboxes `provided` and `required` are added. These mailboxes contain the names of services that are provided to and required from the ambient by an actor, respectively. These names of services determine what part of the environment is reified. The environment is reified through the introduction of the `joined` and `disjoined` mailboxes. These two mailboxes are transparently updated by the actor system and contain the actor addresses of ambient resources that have appeared and disappeared, respectively, from the ambient of a device. As such they reveal the environmental context to the actors.

4.5.1 Simple Ambient Actor Language

The ambient actor language is an extension of the call-by-value lambda calculus that contains standard actor primitives `send`, `become` and `letactor`, which are discussed in section 4.2.1. However, similar to the standard actor model the `letactor` primitive has been decomposed into two primitives `newadr` and `initbeh`. The former creates a new actor address and the latter initialized the behavior for this actor address. Next to the standard actor primitives we add a number of primitives to manipulate mailboxes:

- `messages(e)` returns the set of messages of mailbox `e`.
- `add(mbxName, e)` adds a message `e` to the mailbox with name `mbxName`. A message added to a mailbox that does not exist creates the mailbox.
- `delete(mbxName, e)` deletes a message `e` from the mailbox with name `mbxName`

In the following subsections we define the operational semantics of these operations as an extension of the standard actor model. These operational semantics are defined as transitions of configurations.

4.5.2 Messages and Mailbox Associations

In this part we define a number of sets related to the representation of messages and mailboxes in the ambient actor model. We take as given countable sets $\mathbb{A}\mathtt{t}$ (atoms) and $\mathbb{X}$ (variables including actor mail addresses).

Definition 9 ($\mathbb{V}$ $\mathbb{E}$) *: The set of values, $\mathbb{V}$, the set of expressions $\mathbb{E}$, and, the set of actor states, $\mathbb{A}s$ are defined inductively:*

$$\mathbb{V} = \mathbb{A}t \cup \mathbb{X} \cup \lambda\mathbb{X}.\mathbb{E} \cup pr(\mathbb{V}, \mathbb{V})$$
$$\mathbb{E} = \mathbb{V} \cup app(\mathbb{E}, \mathbb{E}) \cup \mathbb{F}_n(\mathbb{E}^n) \text{ where } \mathbb{F}_n(\mathbb{E}^n) \text{ is all arity-n primitives.}$$
$$\mathbb{A}s = (?_{\mathbb{X}}) \cup (\mathbb{V}) \cup [\mathbb{E}]$$

Say Y is a set then $\mathbf{P}_\omega[\mathtt{Y}]$ is the set of finite subsets of Y. $\mathbf{M}_\omega[\mathtt{Y}]$ is the set of finite multi-sets with elements in Y. $Y_0 \xrightarrow{\mathtt{f}} Y_1$ is the set of finite maps from Y_0 to Y_1. $Dom(f)$ be the domain of f.

A message is represented as a nested pair of value expressions, this is in contrast with the message representation as defined in [AMST97] (where a message was denoted with $< b \Leftarrow cv >$). By representing the messages as a pair of values the message becomes a first class value in the actor language. This will prove useful to manipulate the mailboxes. Another difference with the standard actor model is that the message includes the sending actor (called the source). To make a clear distinction in the definitions between messages and other pair values, we will identify a pair that is used as a message with $b \overset{a}{\Leftarrow} cv$.

Definition 10 (Messages ($\mathbb{M}$))

$$\mathbb{M} = \{pr(a, pr(b, cv)) \in \mathbb{V} \mid a, b, cv \in \mathbb{V}\}$$

A message is a nested pair (pr) of:

- a, the actor address of the source actor.
- b, the actor address of the target actor.
- cv, a communicable value, constructed from atoms and actor addresses, but not containing closures.

In the spirit of dynamic typing (as in [AMST97]) we do not restrict the target of the message to the set of actor addresses, the correctness is checked in the rules that define the operational semantics.

In the ambient actor model a message can be associated with multiple mailboxes. To denote these mailbox associations in the actor model we introduce the following set:

Definition 11 (Mailbox Associations ($\mathbb{mB}$))

$$\mathbb{mB} = \{\beta \in \mathbb{X} \xrightarrow{f} (\mathbb{S} \xrightarrow{f} \mathbb{V}) \mid ct \in \mathbb{V}, a \in \mathbb{X}, mbx \in \mathbb{S}\}$$

$\mathbb{S}$ is the set of identifiers for mailboxes, $\mathbb{S} \subset \mathbb{A}\mathtt{t}$. The set of mailbox associations is a mapping from an actor mail address to a mapping of the names of its mailboxes to their contents. To increase the readability, mappings of β will be written as $< ct \mid mbx_a >$ with $a \in Dom(\beta)$ and $\beta(a) = \delta$. Furthermore,

$mbx \in Dom(\delta)$ and $\delta(mbx) = ct$. $ct \in \mathbb{V}$ denotes the content associated with mailbox mbx_a. The name of the mailbox is written as the identifier $mbx \in \mathbb{S}$, subscripted with the actor address $a \in \mathbb{X}$ to which the mailbox belongs. E.g., $inbox_b$ denotes the `inbox` of actor b. Typically, messages are associated with a mailbox, but other value types can also be associated with a mailbox.

4.5.3 Actor Configurations

The operational semantics of the model itself is based on actor-configurations and reduction rules defined on these configurations. Conceptually, an actor configuration can be perceived as the runtime state of an actor system (discussed in section 4.2.2). Such an actor system runs on any computational device, such as a mobile phone or desktop.

Definition 12 (Actor Configurations ($\mathbb{K}$))

$$\langle\!\langle \alpha \mid \mu \rangle\!\rangle^{\rho}_{\chi}$$

where ρ, $\chi \in \mathbf{P}_{\omega}[\mathbb{X}], \alpha \in \mathbb{X} \xrightarrow{f} \mathbb{A}s$, *and* $\mu \in \mathbf{M}_{\omega}[\mathtt{m}\mathbb{B}]$

An actor configuration contains:

- the state of the actors in a configuration is given by an actor map α. Such an actor map is a finite map from actor addresses to actor states. Each actor state is one of
  - $(?_a)$ uninitialized actor state created by an actor with address a
  - (b) actor state ready to accept a message where b is its behavior represented by a closure
  - $[e]$ actor in a busy state executing expression e. e is either a value expression or a reduction context `R` filled with a redex r (written as `R`$[r]$). The reduction context is used to describe internal transitions while a message is being evaluated by the λ-function associated with the behavior. The current expression that is evaluated is decomposed into a reduction context with a unique hole. For the formal elaboration on reduction contexts we refer to the standard actor model [AMST97]. Suffice it to say that here that `R` in the following definitions ranges over the reduction contexts. The redexes that are not actor-specific expressions, namely the purely functional fragment of the language, are inherited from the operational semantics of the standard actor model. The redexes related to the actor operations are `newadr()`, `init`(a, e), `become`(v), `send`(v_0, v_1), `add`(mbx, ct), `delete`(mbx, ct) and `messages`(mbx).

  Each mapping of an actor address to an actor state is subscripted by their actor address. E.g. $(?_a)_c$ denotes an uninitialized actor c that was created by actor a.

- μ, a multi-set of mailbox associations.
- ρ, receptionists, the actor addresses from this configuration that are accessible from other actor configurations

- χ, external actors, the addresses of actors from other actor configurations that can be accessed from this actor configuration.

It is required that all actor configurations satisfy the following well-formedness constraints (A=Dom(α)):

1. $\rho \subseteq A$ and $A \cap \chi = \emptyset$,
2. if $\alpha(a) = (?_{a'})$, then $a' \in A$,
3. if $a \in A$, then $\mathrm{FV}(\alpha(a)) \subseteq A \cup \chi$,
4. if $< ct \mid mbx_a > \in \mu$, then $a \in A$
5. if $< ct \mid mbx_a > \in \mu$ then $\mathrm{FV}(ct) \subseteq A \cup \chi$

FV(e) is the set of all free variables encountered in an expression e as defined by Agha et al [AMST97].

The fourth constraint is a new constraint and denotes that each mailbox in an actor configuration should be owned by an actor from the actor configuration.

4.5.4 Operational Semantics of Actor Configurations

Now that we have defined the necessary sets involved in the formalization of the operational semantics of the ambient actor model we can define the actual operational semantics. The operational semantics are defined as reduction rules on actor configurations. Conceptually, such a rule can be regarded as an evaluation step of an actor system. Each rule contains a label l that consists of a tag indicating its name and a set of parameters. In all cases, except for the i/o transitions (with tags *local*, *in*, *out*, *ack*, *join*, *disjoin*), the first parameter names the *focus* actor of the transition.

As in the paper of Agha et al. [AMST97] we use the following notation for maps: if the mapping $\alpha'(a) = (b)$ and if α differs from α' in that a is omitted from its domain then we write α' as $\alpha, (b)_a$ such that the focus is on the state of actor a. We follow the same convention for other maps with actor addresses in their domain, such as mailbox associations.

In our model the transitions ($\mapsto$) are extended with an environmental context set τ. The set τ contains the actor configurations that are available (in the communication range of the actor configuration on which the transition is defined) while the reduction is performed. The introduction of this set is important to reify the notion of environmental context in our extended model. Below we explain and discuss the different rules.

Definition 13 ($\underset{\tau}{\mapsto}$) $\tau \in \mathbf{M}_\omega[\mathbb{K}]$

$< \texttt{fun} : a >$

$$e \overset{\lambda}{\mapsto} Dom(\alpha) \cup \{a\}\ e' \Rightarrow \langle\!\langle \alpha, [e]_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, [e']_a \mid \mu \rangle\!\rangle^{\rho}_{\chi}$$

$< \texttt{new} : a, a' >$

$$\langle\!\langle \alpha, \texttt{[R[newadr()]]}_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, \texttt{[R[a']]}_a, (?_a)_{a'} \mid \mu \rangle\!\rangle^{\rho}_{\chi} \qquad a' fresh$$

$< \texttt{init} : a, a' >$

$$\langle\!\langle \alpha, \texttt{[R[init}(a', v)\texttt{]]}_a, (?_a)_{a'} \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, \texttt{[R[nil]]}_a, v_{a'} \mid \mu \rangle\!\rangle^{\rho}_{\chi}$$

$< \texttt{become}\ : a, a' >$

$$\langle\!\langle \alpha, \texttt{[R[become}(v)\texttt{]]}_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, v_a \mid \mu \rangle\!\rangle^{\rho}_{\chi}$$

$< \texttt{send}\ : a, m >$

$$\langle\!\langle \alpha, \texttt{[R[send}(v_0, v_1)\texttt{]]}_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, \texttt{[R[nil]]}_a \mid \mu, m \rangle\!\rangle^{\rho}_{\chi}$$
$$\textit{with } m =< v_0 \overset{a}{\Leftarrow} v_1 \mid \textit{outbox}_a >$$

$< \texttt{local}\ : m >$

$$\langle\!\langle \alpha \mid \mu, m \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha \mid \mu, M \rangle\!\rangle^{\rho}_{\chi}$$
$$\textit{with } m =< b \overset{a}{\Leftarrow} cv \mid \textit{outbox}_a >$$
$$\textit{and } M = \{< b \overset{a}{\Leftarrow} cv \mid \textit{sentbox}_a >, < b \overset{a}{\Leftarrow} cv \mid \textit{inbox}_b >\}$$
$$\textit{if } a, b \in Dom(\alpha) \textit{ and } x = a \vee b \textit{ then } \nexists\alpha(x) = [g] \textit{ with } g \in \mathbb{A}s$$

$< \texttt{out}\ : m >$

$$\langle\!\langle \alpha \mid \mu, m \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha \mid \mu \rangle\!\rangle^{\rho \cup \{a\} \cup (FV(cv) \cap Dom(\alpha))}_{\chi}$$
$$\textit{with } m =< b \overset{a}{\Leftarrow} cv \mid \textit{outbox}_a > \textit{ if } b \in \chi, a \in Dom(\alpha) \textit{ and}$$
$$\nexists\alpha(a) = [g] \textit{ with } g \in \mathbb{A}s$$

$< \texttt{in}\ : m >$

$$\langle\!\langle \alpha \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha \mid \mu, m \rangle\!\rangle^{\rho}_{\chi \cup \{a\} \cup (FV(cv) - Dom(\alpha))}$$
$$\textit{with } m =< b \overset{a}{\Leftarrow} cv \mid \textit{inbox}_b >,\ b \in \rho \textit{ and } FV(cv) \cap Dom(\alpha) \subseteq \rho,$$
$$\textit{if } \nexists\alpha(b) = [g] \textit{ with } g \in \mathbb{A}s$$

$< \texttt{ack}\ : m >$

$$\langle\!\langle \alpha \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha \mid \mu, m \rangle\!\rangle^{\rho}_{\chi}$$
$$\textit{with } m =< b \overset{a}{\Leftarrow} cv \mid \textit{sentbox}_a >,\ FV(cv) \cap Dom(\alpha) \subseteq \rho,$$
$$\textit{if } b \in \chi, a \in Dom(\alpha) \textit{ and } \nexists\alpha(a) = [g] \textit{ with } g \in \mathbb{A}s$$

$< \texttt{rcv}\ : a, m >$

$$\langle\!\langle \alpha, (v)_a \mid \mu, m \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, \texttt{[app(}v, a \overset{b}{\Leftarrow} cv\texttt{)]}_a \mid \mu, m' \rangle\!\rangle^{\rho}_{\chi}$$
$$\textit{with } m =< a \overset{b}{\Leftarrow} cv \mid \textit{inbox}_a > \textit{ and } m' =< a \overset{b}{\Leftarrow} cv \mid \textit{rcvbox}_a >$$

$< \texttt{messages}\ : a, mbx >$

$$\langle\!\langle \alpha, \texttt{[R[messages}(mbx)\texttt{]]}_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, \texttt{[R[}(ct_1, \ldots, ct_n)\texttt{]]} \mid \mu \rangle\!\rangle^{\rho}_{\chi}$$
$$\textit{with } ct_i \in \{ct \mid < ct \mid mbx_a > \in \mu\}$$

$< \texttt{add}\ : a, mbx, ct >$

$$\langle\!\langle \alpha, \texttt{[R[add}(mbx, ct)\texttt{]]}_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \underset{\tau}{\mapsto} \langle\!\langle \alpha, \texttt{[R[nil]]}_a \mid \mu, m \rangle\!\rangle^{\rho}_{\chi}$$
$$\textit{with } m =< ct \mid mbx_a >$$

$< \texttt{delete} : a, mbx, ct >$

$\langle\!\langle \alpha, [\texttt{R}[\texttt{delete}(mbx, ct)]]_a \mid \mu \rangle\!\rangle^{\rho}_{\chi} \overset{\tau}{\mapsto} \langle\!\langle \alpha, [\texttt{R}[\texttt{nil}]]_a \mid \mu' \rangle\!\rangle^{\rho}_{\chi}$
with $\mu' = \mu \backslash \{< ct \mid mbx_a >\}$

$< \texttt{join} >$

$\langle\!\langle \alpha_0 \mid \mu_0 \rangle\!\rangle^{\rho_0}_{\chi_0} \overset{\tau}{\mapsto} \langle\!\langle \alpha_0 \mid \mu_0, M \rangle\!\rangle^{\rho_0}_{\chi_0 \cup \{a\}}$
if $\exists \kappa \in \tau$ *with* $\kappa = \langle\!\langle \alpha_1 \mid \mu_1 \rangle\!\rangle^{\rho_1}_{\chi_1}$ *and* $\nexists \alpha_0(b) = [g]$ *with* $g \in \mathbb{A}s$ *and*
$M = \{< pr(a, cv) \mid joined_b >, < b \overset{b}{\Leftarrow} \texttt{join} \mid inbox_b > \mid < cv \mid required_b > \in \mu_0 \wedge < cv \mid provided_a > \in \mu_1\}$

$< \texttt{disjoin} >$

$\langle\!\langle \alpha_0 \mid \mu_0 \rangle\!\rangle^{\rho_0}_{\chi_0} \overset{\tau}{\mapsto} \langle\!\langle \alpha_0 \mid \mu_0 \backslash T, M \rangle\!\rangle^{\rho_0}_{\chi_0}$
if $\nexists \kappa \in \tau$ *with* $\kappa = \langle\!\langle \alpha_1 \mid \mu_1 \rangle\!\rangle^{\rho_1}_{\chi_1}$ *and* $\nexists \alpha_0(b) = [g]$ *with* $g \in \mathbb{A}s$ *and*
$M = \{< pr(a, cv) \mid disjoined_b >, < b \overset{b}{\Leftarrow} \texttt{disjoin} \mid inbox_b > \mid$
$< pr(a, cv) \mid joined_b > \in \mu_0 \wedge a \in Dom(\alpha_1)\}$
$T = \{< pr(a, cv) \mid joined_b > \mid < pr(a, cv) \mid joined_b > \in \mu_0 \wedge a \in Dom(\alpha_1)\}$

Basic Actor Operations

The first three reduction rules below remain unchanged with regard to the actor model. The `become` on the other hand had to be modified to ensure correct semantics with respect to the first-class mailboxes.

- The $< fun >$ rule above delegates the purely functional expressions used in the actor program to the functional redexes. The functional redex contains reduction rules for function calls, cons-cell manipulation, branch-testing, type-testing and equality. For the exact definition of these reduction rules we refer the reader to [AMST97].

- The semantics of the `letactor` primitive is formalized by two rules, $< new >$ and $< init >$. The $< new >$ rule is used to create a new actor with address *a'*. The new actor is not initialized after this reduction. We say that a variable is *fresh* with respect to a context of use if it does not occur free or bound in any syntactic entity. The new uninitialized actor is denoted with $(?_a)_{a'}$.

- With the $< init >$ rule a new actor is initialized with behavior v. Only the actor that created the actor *a'* can initialize it.

- With the $< become >$ rule the actor can change its state and behavior, similar to the `become` rule in the standard actor model. However, in the rule defined by Agha et al. [AMST97] the expressions evaluated after a `become` will be further reduced in the context of a new anonymous actor. It is this anonymous actor that introduces the intra-object concurrency in the actor model.

  In the changes we have made, this remaining expression is not further reduced, changing its semantics similar to a `break` instruction found in many

programming languages. The reason we removed intra-object concurrency is to ensure mailbox manipulations have correct semantics. Indeed, if the remaining expression is reduced in the context of an anonymous actor and mailboxes would be manipulated then the mailboxes of the anonymous actor would be manipulated instead of the actor that started to process the message. This would lead to awkward semantics. An alternative to this solution would be to let an anonymous actor manipulate the mailboxes of the actor that spawned it. However, this would introduce race conditions on the mailboxes. The topic of safe access to mailboxes is further discussed below in section 4.5.5.

Part of the parallelism found in the actor model stems from the asynchronous message passing that is used as we explained in section 2.4.4. In fact, many contemporary implementations of programming languages based on the actor model have made a similar tradeoff. In languages such as Salsa [VA01] and E [Mil04] the `become` operation has been replaced by assignments that are used throughout a method body. In these languages the intra-object concurrency has also been removed to ensure that no race conditions occur on the internal state of the actor. It is true that removing the intra-object concurrency reduces the massive parallelism that was found in the actor model, but we do not believe that the change influences the workability of the actor model for mobile networks.

Communication Rules

The remainder of the rules have been adapted to include the notion of mailboxes:

- The $< send >$ rule mildly differs from the *send* rule found in the actor model. The new rule reduces the send operation to placing the message in the `outbox` of the actor in which the send operation is reduced.
- $< local >$ is a new rule that was added to model local communication between actors. If the message can be delivered locally (within the same actor configuration), it is placed both in the target its `inbox`, and the `sentbox` of the sender. The rule is defined such that communication can only occur when the two actors involved in the communication are not in a busy state. Similar conditions are also specified for the other i/o-transitions. These conditions are necessary to preserve the model from race conditions. This will be discussed in section 4.5.5.
- $< out >$ The out reduction rule is used at the sending side for messages that cannot be delivered locally, to transmit a message to another actor configuration. Similar to the original model, the set of receptionists is expanded with the local actor addresses that were communicated in the message. The outgoing message is removed from the `outbox`.
- $< in >$ an actor configuration receives a message from an external actor that runs on another actor system. In this situation, the message is placed in the `inbox` of the target actor.
- $< ack >$ an actor configuration receives an acknowledgment for a message it send and places that message in the `sentbox` of the sending actor. This allows the actor to verify which messages have actually been sent.

- $< rcv >$ When a message is available in the `inbox` of an actor, it can be received by the actor and when it is processed by the actor, it is moved to the *rcv* mailbox. As a result, an actor has a history of the messages that it processed. This proves to be useful for determining the communication state of an actor as is shown in the examples in section 4.6.

Mailbox Manipulation

$< messages >, < add >, < delete >$ Some reduction rules have been added to manipulate and inspect the mailboxes from within the actor language. With the $< messages >$ rule one can access the content of a mailbox. The $< add >$ rule creates a mailbox when it does not exist, if the mailbox exists, the content will be added to the mailbox. The $< delete >$ rule delete a message from a mailbox, when the last message of a mailbox has been removed, the mailbox itself is removed. The above reduction rules allow actors to manage mailboxes explicitly. Note that there is no rule in which a message automatically disappears from the system. This means that memory management will have to be handled manually by the programmer. This is because it depends on the semantics of the program whether a message has become irrelevant to the program. For example, when a certain task has completed and its associated messages are not relevant anymore.

Handling Environmental Contexts

AmOP applications need to have access to the reified environmental context (discussed in section 3.5) so that they can address ambient resources (discussed in section 2.3). The reification of the environmental context is supported with two reduction rules: $< join >$ and $< disjoin >$. When two devices are in the communication range of one another, their actor systems will automatically "join". They disjoin when they leave each others communication range. Actors are usually interested in a specific resource from the ambient and are only interested in the (dis)appearance of these resources. To this end, four extra mailboxes have been added for each actor: `provided`, `required`, `joined` and `disjoined`. The mailboxes `provided` and `required` are used to let an actor specify an abstract description of what kind of behavior it provides or requires, this abstract description is called a *pattern*. The pattern is specified in the model as a communicable value. When a pattern in the `provided` and `required` mailboxes of different actors match, then the actor that required the pattern will be notified. This notification happens through the use of the `joined` and `disjoined` mailboxes. Thus, the `joined` and `disjoined` mailboxes keep track of the relevant actors, specified through the use of the `provided` and `required` mailboxes, that are in communication range. This mechanism is defined in the model through the $< join >$ and $< disjoin >$ rules:

- $< join >$ when two actor configurations come in the communication range of one another then every actor b that requires a certain pattern cv, which has become available in another actor configuration κ that is in communication range, will be informed of this by receiving a "join" message in its `inbox`. Also, for every matching pair of required-provided patterns, the corresponding `joined` mailbox is updated. In the `joined` mailbox, a special kind of message is stored, called a *resolution*. A resolution contains

a) the *pattern* (cv) that has been matched and b) a *provider* actor a who provides the service represented by the pattern. Hence, the resolutions found in the mailbox of an actor specify the actor addresses of ambient resources that matched the pattern.

- The $< disjoin >$ rule specifies the semantics of two actor configurations that leave each others communication range. Every actor that is aware of another joined actor that has left the communication range, will be informed of the disjoin. Once an actor is informed the corresponding resolution is removed from the `joined` mailbox. Actors that have removed the matching messages from their `joined` mailbox will not be informed.

The join and disjoin operations are not the inverse of one another. After joining and disjoining two actor configurations, the state of the involved actor configurations is not necessarily the same as before the join operation. This is due to the fact that for every join or disjoin a number of messages are sent, which might influence the behavior of the involved actors.

4.5.5 Concurrency Issues with Mailboxes

When scrutinizing the ambient actor model, one has to investigate whether if there are concurrency issues involved with the reification of the mailboxes. For example, a possible race condition is an actor that deletes a message from its `outbox` at the moment it is transferred to the target actor its `inbox`. The operational semantics of the ambient actor model exhibit two mailbox properties that are important to avoid race conditions on the mailboxes of an actor.

1. *Mailbox Privacy*
   Each mailbox has a unique name within an actor. A mailbox is associated with exactly one actor and an actor cannot communicate a reference to one of its mailboxes.[3] Hence, mailboxes are never shared among multiple actors. This is called the *mailbox privacy* property.

2. *Serial Mailbox Access*
   In the ambient actor model a mailbox is manipulated by two different entities: the actor owning the mailbox and the actor system which updates mailboxes when communication events occur, for example when a message is transmitted. The operational semantics of the ambient actor model is defined in such a way that the manipulation of mailboxes by these two entities cannot occur concurrently. Indeed, the rules where the actor system manipulates mailboxes as a result of a communication event have the explicit condition that the actor whose mailboxes are being manipulated is not in a busy state. Hence, while an actor is processing a message its mailboxes can only be changed by itself, not by the actor system. Messages that the actor system cannot send at that time remain in the `out` mailboxes of the corresponding actors until they can be transmitted. The characteristic that only one entity can manipulate a mailbox at a time is called the *serial mailbox access* property.

[3] However, it is possible to communicate the name of a mailbox, but this name refers to the local mailbox of the receiving actor and not to the mailbox of the sending actor.

Both the mailbox privacy and the serial mailbox access properties are important in the context of implementations based on this model, because they preserve the encapsulation of the actors and avoid race conditions on mailboxes. These properties will have to be guaranteed by concrete implementations of the model.

4.5.6 Summary and Discussion

In section 4.3 we have discussed the actor model in the context of the ambient-oriented programming characteristics defined in chapter 3 and, although the model provides *non-blocking* communication primitives, we concluded that the model lacks *reified communication traces* and *reified environmental context*. The lack of support for these two AmOP criteria induced that actors are unaware of the (dis)appearance of ambient resources and that the actor model could not support volatile connections sufficiently. The ambient actor model resolves these restrictions with the introduction of *explicit* mailboxes. The use of such mailboxes is twofold: making the communication state of an actor explicit and allowing for *ambient acquaintance management*. Both uses are detailed below.

Communication State

When scrutinising the communication structure of the actor model, we can distinguish between four types of messages. The first type of messages are those an actor received but still needs to process. A second type of messages are those the actor has sent but that have not yet been transmitted. Third, there are messages that an actor has received and processed. Finally, there are the messages that an actor has sent and transmitted. Together, these four types of messages describe the complete communication trace of an actor over time.

In contrast to the standard actor model, where every actor merely has an implicit message set for accumulating incoming and outgoing messages, the ambient actor model allows clear distinction of these four types of messages by introducing four explicit mailboxes. The messages of the first type are put in the mailbox `inbox`, the second type of messages are put in the mailbox `outbox`. If an actor receives a message, then that message will be put in the mailbox `inbox`, waiting to be processed by that actor. When a message is sent by an actor it is put in its mailbox `outbox`, waiting to be transmitted to the recipient of that message. Both mailboxes `inbox` and `outbox` are implicitly present in the actor model and enable the non-blocking communication primitives, which are a necessary characteristic for the ambient-oriented programming paradigm as argued in section 3.3.

In addition to the mailboxes `inbox` and `outbox` there are two more mailboxes, `rcvbox` and `sentbox`, for the third and fourth type of messages respectively. In the ambient actor model, when a message is processed it is moved from the mailbox `inbox` to the mailbox `rcvbox` and when a message is actually transmitted to another actor, then the message is moved from the mailbox `outbox` to the mailbox `sentbox`.

Conceptually, the mailboxes `rcvbox` and `sentbox` allow one to have a peek in the past of the communication history of an actor. Note that the mailboxes `inbox` and `outbox` of the actor represent its continuation, because these two mailboxes contain the messages it will process and transmit in the future. Hence, through the introduction of these four explicit mailboxes we have a gate to the

past and the future of the actor's state of communication, which enables the reified communication traces that were argued in section 3.4.

The ambient actor model provides explicit control over the communication state of an actor through mailbox manipulations. Apart from the four mailboxes that control the state of communication, every actor can create custom mailboxes. Messages can reside in multiple mailboxes at the same time. The status of the delivery of a message can be monitored and altered by accessing the appropriate mailbox. For example, by removing a message from the mailbox `out` we can stop the message from being delivered. Hence, by giving access to the mailboxes, first-class continuations are attained. The mailboxes `in` and `out` not only allow one to have a peek in the future computation and communication of the actor, but even to manipulate it. For example, we could remove a message from the mailbox `in` and thereby prevent it from being processed by the actor.

Ambient Acquaintance Management

In section 3.5 we argued that a form of ambient acquaintance management should be possible in ambient-oriented programming languages. In the AAM, distributed naming is available via a pattern-based lookup mechanism. A pattern is an abstract description of a set of actors and is specified by a communicable value. An actor that wants to search for certain other actors in its ambient places a corresponding pattern in its mailbox `required`. Conversely, when an actor wants to make itself available for other actors it places a pattern with a description of itself in its mailbox `provided`. In the former case the actor is said to *require* a pattern, while in the latter case the actor is said to *provide* a pattern. Multiple patterns can be added to a mailbox such that an actor can require or provide multiple patterns simultaneously. A pattern can also be removed from either mailboxes at any time when the actor no longer requires or provides a certain pattern.

When two or more actors enter one another's communication range and have a corresponding pattern in their mailboxes, the mailbox `joined` of the actor that required the pattern is updated with a *resolution*. Such a resolution is a pair consisting of the pattern and a reference to the actor who provided the pattern. Conversely, when two actors with a corresponding pattern in their mailboxes are pulled out of communication range, the resolution is moved from the mailbox `joined` to the mailbox `disjoined`. This mechanism allows actors not only to detect new resources in its ambient, but also to detect when actors have disappeared from the ambient. Through this mechanism an actor can manage the acquaintances it encounters in its ambient, which is a characteristic required for ambient-oriented programming languages as discussed in section 3.5

4.6 Examples

Now that we have defined the operational semantics of the ambient actor model we show that it is useful in the context of mobile networks by means of two examples. The examples are defined with actor code based on the semantics of the ambient actor model from the previous section. The first example shows how anonymous communication can be expressed. In the second example we

elaborate on a meeting scheduler application for use in a mobile ad-hoc network.

In the examples below we use the convention that functions prefixed with `mk` create the respective messages and functions that end with a "?" are predicates. For example, `mkPrint` is a function that creates the `print` message and `print?` is a function that returns true if its argument is a `print` message.

4.6.1 Pattern-Based Communication

In section 3.5 we discuss that distributed naming is a good abstraction to communicate with resources in the ambient for which the address is unknown. For example, an actor that wants to print a file on a printer first needs to locate a suitable printer in the ambient and then communicate with it. With a distributed naming scheme both actions can be abstracted in a single communication instruction `psend` that allows an actor to be named based on its properties rather than on a specific address. Below is the definition of an actor using the `psend` abstraction to print a file from the moment it comes into communication range of an actor that provides a printing service.

$B_{Customer}$ = λfile.λm.
```
    seq(psend('printer@300dpi, mkPrint(file)),
        become(handleJoin))
```

In the ambient actor model the addresses of ambient resources can be retrieved based on the pattern through the mailboxes that reify the environmental context. Hence, an actor can be described using a pattern that embeds the type information [KB02] or more semantic information about the service. We define a new communication primitive *psend* that takes two parameters: a service description *pattern* of the required actor and the message that is to be sent.

```
psend = λpattern.λmsg.
   seq(add('required, pattern),
       add('pending, msg))
```

We add the description `pattern` of the required actor to the `required` mailbox. This way the actor will be notified when the actor configuration joins with another actor configuration providing this pattern. The message that is to be communicated is placed in a custom mailbox `pending`. Hence, the `pending` mailbox can be regarded is a special `outbox` for messages that have a pattern as destination instead of an actor address.

The `handleJoin` definition listens for `join` messages that indicate that the `joined` mailbox has been changed. In such an event it runs through the resolutions in the `joined` mailbox. Each time a pattern that corresponds with the target of the messages in `pending` mailbox is found, the message is sent to the provider of the pattern and removed from the `pending` mailbox.

```
handleJoin = λmsg.
   if(join?(msg),
       for-each(λresolution.
           for-each(λpmsg.
               if(eq?(target(pmsg), pattern(resolution)),
                  seq(send(provider(resolution), pmsg),
                      delete('pending, pmsg))),
```

```
            messages('pending)),
        messages('joined)))
```

This first example has shown that the ambient actor model contains the necessary primitives that can be used to build more complex distributed naming schemes. The distributed naming scheme is realized through an abstraction that unifies message sending with the discovery of services such that actors can send messages based on a pattern specification rather than their explicit mail address. This abstraction is implemented based on the mailboxes that reify the environmental context and confirms the need for this AmOP criterion.

4.6.2 Meeting Scheduler

Suppose we have a calendar application (running on a mobile device such as a PDA or mobile phone) that helps us to schedule and remind us of appointments with a group of acquaintances. Each mobile device executes the agenda application and a request for a meeting can be initiated at any point in time, irrespective of whether the agenda applications of the acquaintances are available for communication. This kind of behavior is necessary to support the autonomous mobility of the users and the fact that some mobile devices may not be available due to volatile connections. For that reason, the application has to deal with volatile connections and the application may not rely on a central server architecture, which are two hardware phenomena we discussed in section 2.3. We assume that the mobile devices at some point in time will be in the communication range of one another.

The agenda application schedules a meeting in two steps.

1. It tries to make a reservation in the agenda of the participants of the meeting.
2. It confirms the reservation in the agenda of each participant if all individual reservations were successfully reserved.

If the reservation fails at some point, then all individual reservations that were made on other agendas are removed. Each agenda application comprises two actors that are described below.

Agenda Actor

Each agenda is initialized with the e-mail address of the agenda's owner. The e-mail address is combined with the type information of the application to form a pattern, which is returned by the `mkPattern` function, and uniquely identifies the agendas. This pattern is added to the `provided` mailbox such that the presence of each individual agenda of the participants can be detected in the ambient.

```
B_InitAgenda = λemail.λm.
   seq(add('provided, mkPattern(email)),
       become(B_FreeAgenda()))
```

For the sake of the argument, we have chosen to represent the agenda as a single slot that is available for meetings. The slot has three states: `FreeSlot`,

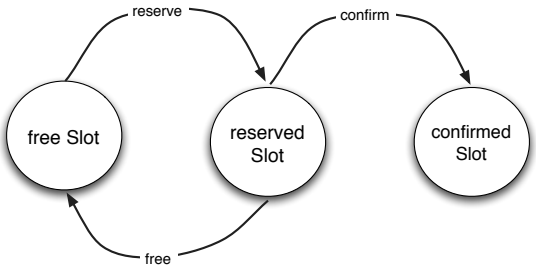

Figure 4.2: State Chart of Agenda Behavior

`ReservedSlot` and `ConfirmedSlot`. A slot understands three messages, `free`, `reserve` and `confirm` and corresponding to the message it receives the slot moves into another state:

- `free`: when this message is received and the slot is in a *reserved* state then it becomes available for reservation. This message is used to undo a reservation.
- `reserve`: when the state of the slot is *free* and this message is received then the slot moves to the *reserved* state.
- `confirm`: when the state of the slot is *reserved* and a `confirm` message is received then the slot moves to the *confirmed* state.

The state chart for the agenda's behavior is shown in figure 4.2. Note that we did not consider erroneous state transitions which should notify the sender of the message. The code below shows the implementation of the slot behaviors.

```
B_FreeSlot = rec(λb.λm.
    if(free?(m),
       become(b))
    if(reserve?(m),
       seq(send(sender(m), mkReserveAnswer(session(m), #true)),
           become(B_ReservedSlot(session(m))))))

B_ReservedSlot = rec(λb.λid.λm.
    if(and(free?(m), eq?(id, session(m))),
       become(B_FreeSlot()))
    if(reserve?(m),
       seq(send(sender(m), mkReserveAnswer(session(m), #false)),
           become(b(id))))
    if(and(confirm?(m), eq?(id, session(m))),
       become(B_ConfirmedSlot(id))))

B_ConfirmedSlot = rec(λb.λid.λm.
    if(reserve?(m),
       seq(send(sender(m), mkReserveAnswer(session(m), #false)),
           become(b(id)))))
```

Each *reserved* and *confirmed* state has a session id that is used to determine to whom the slot has been assigned. The slot only evolves into the corresponding state if the message contains the right session id.

Scheduler Actor

Each agenda application comes with a scheduler agent. This agent is responsible for contacting the agenda actors to schedule the meeting. In the actor definitions of the scheduler implementation below we use four helper functions:

- a `filter` function that uses a predicate to filter elements in a list[4]

```
filter = rec(λb.λpredicate.λlist.
  if(empty?(list),
     emptyList,
     if(predicate(car(list)),
        cons(car(list), b(predicate, cdr(list))),
        b(predicate, cdr(list)))))
```

- a `map` function that returns a transformed list and takes two parameters: a function that transforms elements and a list that is to be transformed is defined as in standard Scheme implementations [AS96].

- `msend` that allows sending a message to a list of actor addresses or actors described with pattern descriptions (such as in the previous example)

```
msend = λtargets.λm.
  for-each(λtarget.
              if(actorAddress?(target),
                 send(target, m),
                 psend(target, m)),
           targets)
```

- `madd` that allows adding a list of messages to a mailbox

```
madd = λmbx.λitems.for-each(λitem.add(mbx,item), items)
```

- `mdelete` that allows to delete a list of messages from a mailbox

```
mdelete = λmbx.λitems.for-each(λitem.delete(mbx,item), items)
```

The scheduling agent behavior is initialized as $B_{InitScheduleAgent}$. The scheduler agent has an `id` that is used to identify its session.

$B_{InitScheduleAgent}$ `= rec(λb.λid.λm.`
```
    if(schedule?(m),
       seq(msend(participants(m), mkReserve(id)),
           become(B_ReserveScheduleAgent
                    (id, participants(m), sender(m)))))
```

[4] In the code below we use LISP terminology for our function related to lists. A pair is created with the function `cons`. The function `car` returns the first element of the pair, while the `cdr` function returns the second element. A list of elements is represented as a nesting of pairs. e.g. (1, 2, 3) is represented as (1, (2, (3, '())))

The schedule agent can be requested to schedule a meeting with a group of participants by sending it the message `schedule`. This message contains the unique patterns (created with `mkPattern`) that identify the agenda of each participant. These patterns are retrieved from the message with the function `participants`. When such a request is received the agent sends out `reserve` messages to the agenda actors of the participants and the scheduler evolves into the $B_{ReserveScheduleAgent}$ state.

```
B_ReserveScheduleAgent = rec(λb.λid.λparticipants.λcustomer.λm.
    if(and(reserveAnswer?(m), eq?(id, session(m))),
       if(success?(m),
          if(eq?(map(sender,
                     filter(reserveAnswer?, messages('rcvbox)))),
                 participants),
             seq(msend(participants,mkConfirm(id)),
                 become(B_ConfirmScheduleAgent
                          (id, participants, customer)))),
          seq(msend(map(destination,
                        filter(reserve?, messages('sentbox)))),
                    mkFree(id)),
              mdelete('outbox,
                      filter(reserve?, messages('outbox))),
              send(customer, mkFailed()),
              become(B_InitScheduleAgent(id+1)))),
       seq(handleJoin(m),
           become(b(id, participants, customer)))))
```

When handling a `reserveAnswer` message we make use of the mailboxes introduced in our model:

- If the reservation was successful the `rcvbox` is checked to determine if the scheduling agent has received an answer from all the participants their agendas. Thanks to the reification of the communication traces by the mailboxes there is no need to manually maintain the progress of the meeting scheduler. If all agendas are successfully reserved, then the `ScheduleAgent` actor sends `confirm` messages to all agendas.
- If an individual reservation request fails then the scheduler agent has to free up the slots of the reservations that were successful. The `sentbox` can be used to track to which agenda actors the scheduler agent has already sent reservation request. These actors are sent a `free` message so that they can undo their reservation. Furthermore, the scheduler agent deletes the `reservation` messages from the `outbox` and thereby undoes the communication before it occurred. Next, the scheduler agent also notifies the customer actor that sent the `schedule` message that the meeting could not be scheduled by sending it a `failed` message.

```
B_ConfirmScheduleAgent = rec(λb.λid.λparticipants.λcustomer.λm.
    if(and(disjoin?(m),
           eq?(map(sender, filter(confirm?, messages('sentbox)))),
               participants))
```

```
seq(send(customer, mkSucceeded()),
    become(B_InitScheduleAgent(id+1))),
seq(handleJoin(m),
    become(b(id, participants, customer)))))
```

Each time the `ScheduleAgent` actor disjoins from an agenda actor it checks to see if all `confirm` messages were sent out, again using the `sentbox`. If all `confirm` messages were sent, then the customer actor that sent the `schedule` is notified with a *succeeded* message.

The second example has shown that the mailboxes, which reified the communication traces, introduced in the ambient actor model contain primitives that allow the scheduler agent to deal with the autonomous and concurrent nature of the devices. Indeed, the scheduler agent consults and manipulates its `outbox` and `sentbox` to keep pace with the communication status of the different messages. Note that the implementation above relies on the eventual delivery of messages. More advanced delivery policies can be devised for this application based on the manipulation of mailboxes. For example, a broken connection between applications could be intercepted with a `disjoin` message and unsent reservation requests could be reversed by removing them from the `outbox`.

4.6.3 Discussion

These two examples demonstrate that, through the support of the AmOP criteria, the ambient actor model can deal with the hardware phenomena, which were discussed in section 2.3. However, some of the implementation details indicate that the object model lacks expressiveness. Unfortunately, this lack of expressiveness is inherited from the "object model" of the standard actor model, which we discussed in section 4.3.1. This lack of expressiveness leads to complex code. For example, an actor that uses of the `psend` abstraction must also manually incorporate the `handleJoin` in its behavior such that it understands the `join` messages and can handle them accordingly. In the code, the `reserveScheduleAgent` and the `ConfirmScheduleAgent` behaviors both had to embed this handler. Hence, if abstractions and behaviors become more advanced then the complexity for embedding them also grows. This complicates the introduction of high-level language abstractions that can deal with the hardware phenomena, one of the research goals we set in section 1.1.3.

The introduction of such language abstractions is necessary because the manipulation of mailboxes is low-level. In many respects first-class mailboxes can be regarded as a representation of first-class continuations for the actor model. In sequential languages first-class continuations can be used to implement many high-level abstractions, such as exception handling systems and coroutines. However, unlike most of these high-level abstractions, first-class continuations themselves are cumbersome to use and sometimes difficult to understand. The same observation also holds true for the use of first-class mailboxes. In other words, instead of programming an AmOP application with mailboxes we want to program with high-level abstractions.

Putting the ambient actor model in perspective, it could be regarded as a specification of a low level language for ambient-oriented programming languages.

4.7 Conclusion

In this chapter we extended the operational semantics of the actor model in order to deal with three problems associated with mobile (ad-hoc) networks:

1. How to deal with ambient resources?
   To handle this problem we introduced reduction rules *join and disjoin* that specify what happens when devices come in communication range of one another.

2. How to deal with volatile connections?
   This problem is handled with the introduction of the `inbox`, `rcvbox`, `outbox` and `sentbox` mailboxes that allow one to track and intervene in the communication between different devices.

3. How to deal with the naturally concurrent and autonomous nature of the interactions?
   This is handled through the manipulation of mailboxes that reify the communication traces and through the manipulation of custom mailboxes.

These three problems are handled with the introduction of a single concept in the actor language, namely mailboxes. These mailboxes realize the different AmOP criteria we have distilled in chapter 3 and allowed us to express small examples of code in the formalism that deal with the hardware phenomena we discussed in section 2.3. However, these examples have also shown that the underlying object model lacks expressiveness such that we have come to regard the instruction set found in the ambient actor model as a low level language for ambient-oriented programming languages.

Chapter 5

A Kernel Language for Ambient-Oriented Programming

In the previous chapter we have extended the formal actor model such that it fulfills the ambient-oriented programming characteristics, which we proposed in chapter 3. Unfortunately, the proposed extension of the actor model inherits the low-level nature of the actor model that hampers the introduction of high-level language features that deal with the issues in mobile distributed systems, which is the first research goal we have set in section 1.1.3. Therefore, to overcome this limitation this chapter presents a new language, which is based on the ambient actor model, but provides a richer object model and explicit syntax to better support the implementation of AmOP programming abstractions and applications.

5.1 Introduction

Finding high-level abstractions that facilitate the development of software for mobile networks is difficult because there is little engineering experience about the development of these types of applications. Therefore, the next step is not to design a full-fledged fixed high-level ambient-oriented programming language but instead the goal is to create an extensible ambient-oriented kernel language. A kernel will facilitate experimentation with language features for developing software for mobile networks.

In the previous chapter we have extended the standard actor model such that it can deal with the hardware phenomena, which we discussed in section 2.3. The issues that arise from these phenomena are addressed through the introduction of first-class mailboxes in the standard actor model, which made the ambient actor model support the AmOP criteria (discussed in chapter 3). However, we concluded that the simple ambient actor language defined by the operational semantics was too low-level to introduce high-level language features, which is a requirement to support our experimental approach (discussed in section 1.4.1).

In this chapter we will make a first step in the direction of a language that

supports the introduction of high-level language features by defining an extensible kernel language, called AmbientTalk [DVM+06]. The goal of the AmbientTalk kernel is to incarnate the concurrency and distribution model specified by the ambient actor model as an expressive object-oriented language. In the next section we detail the design decisions we have made for this language. These decisions are regarding the integration of concurrency and distribution into the language. One of the decisions we make is to chose for a double-layered object model, which makes a distinction between passive and active objects. The former object layer, which supports the definition of sequential objects, is detailed in section 5.3. The latter object layer captures both concurrency and distribution aspects and is discussed in section 5.4.

5.2 Design Rationale

In the design of AmbientTalk's object model we have made a number of decisions about the integration of concurrency and distribution aspects. The most important decisions are that we have chosen to reconcile mutable state with concurrency. Hence, we have replaced the actor model's functionally inspired object system for an object system with mutable state. Another important decision is that we have chosen for a double-layered object system that makes a distinction between passive and active objects. Active objects are AmbientTalk's incarnation of actor objects described by the ambient actor model. Finally, we have further chosen active objects as the unit of distribution. These choices are elucidated below.

5.2.1 Reconciling Mutable State with Concurrency

The model of concurrency of AmbientTalk resembles that of ABCL/1 [YBS86] in that it tries to reconcile the imperative style of programming with concurrency. An imperative programming style combined with concurrency can cause race conditions on the data that is shared among multiple processes. Such a style of concurrent programming is often associated with a threads. In such a thread-based model of concurrency it is necessary to preserve the consistency of shared data through the use of locks. Locks allow one to define delimited regions that can be accessed by only one thread at a time. Thread based models have the disadvantage that shared data is mostly implicitly defined such that it is hard to determine where locks have to be placed. If such a place is missed then race conditions can occur. However, race conditions are hard to debug because of the non-deterministic nature of concurrency. On the other hand there is the functional way of dealing with concurrency. In a pure functional language data consistency is a non-issue, because a purely functional program does not change the state of a program. Such programs can be made parallel based on the data-flow of the program. This extremum is impractical because most programs need some form of mutable state.

The actor model [AMST97] reconciles mutable state with a concurrent functional style of programming. As we have seen in the previous chapter, the actor model does not feature an assignment operator. Instead it relies on the *become* operation, which allows an object to change its behavior and state in one single operation. The *become* operation not only changes the behavior and state of

an actor, but also introduces parallelism. Indeed, the new behavior of the actor starts processing the next message concurrently with the expressions succeeding the *become* operation in the context of the old state of an actor. This is an important distinctive characteristic, because the *become* operation introduces intra-object concurrency without the negative consequences of race conditions on the internal state of an actor. The reason that there are no race condition issues on the internal state of an actor is that there is exactly one *become* operation executed when a message is processed. Hence, only one process at a time can change the behavior and state of an actor at a time thereby precluding race conditions on the state of an actor. Unfortunately, the consistency is preserved at the cost of an non-intuitive programming style. The reason for this is that only one state-change is allowed for each message that is processed, which forces one to reorder statements such that all the state changes can be embedded in one operation.

ABCL/1, which used the actor model as a starting base for its design, reintroduced assignments in its object model for concurrency. An active object in ABLC/1 has a single thread of execution and a queue containing messages. The thread continuously processes messages, one at a time, until there are no more messages in the queue or until no appropriate message can be found. Although this model tries to reconcile mutable state with concurrent objects it does not succeed in precluding race conditions on the internal data. The reason for this is that the processing of ordinary messages can be preempted by express mode messages. Such potential race conditions have to be prevented by placing a group of statements that should not be preempted in an `atomic` block.

AmbientTalk also reintroduces the assignments in its object model. However, as opposed to ABCL/1 when a message is processed it always runs to its end before any other message is processed. Languages and middleware based on asynchronous sequential processes (ASP) [CHS04, CH05], which is based on the *wait-by-necessity* scheme [Car89], also process at most one single message at a time. This scheme is based on the transparent use of futures in a distributed setting. If an active object needs access to a future and the future has not been resolved then the active object waits until the future is resolved or an exception is thrown. As a consequence, an active object can be interrupted such that it cannot process another message meanwhile. Hence, ASP-based models conflict with the non-blocking communication criterion of the AmOP paradigm, which was discussed in section 3.3.

Languages such as Salsa [VA01] and E [MTS05, Mil04] also reintroduced assignments in their object model. However, these languages differ from ASP based languages in that active objects cannot be interrupted and always finish executing to their end without being interleaved or interrupted by other active objects. AmbientTalk shares these properties with these two languages.

5.2.2 Double-Layered Object Model

In AmbientTalk we have chosen for a double-layered object model. This double-layered object model distinguishes between *passive objects* and *active objects*. Active objects encapsulate a graph of passive objects. The behavior of the active object is defined by the passive object that is pointed to by the active object.

To avoid having every single object to be equipped with heavyweight concurrency machinery and having every single message to be thought of as a

concurrent one, an object model that distinguishes between active and passive (i.e. ordinary) objects is adopted. This allows programmers to deal with concurrency only when strictly necessary (i.e. when considering semantically concurrent and/or distributed tasks). Since passive objects are not equipped with an execution thread, the "current thread" runs from the sender into the receiver, thereby implementing synchronous message passing. However, it is important to ensure that a passive object is never shared by two different active ones because this easily leads to race conditions. For this reason we have declared the *containment principle*:

> A passive object is contained within exactly one active object.

This principle avoids passive objects from being shared among active objects, thereby removing the source for race conditions on the internal state of objects. Implementing the containment principle implies changing the semantics of some evaluation rules. In principle, each time a passive object is passed from the context of one active object to another e.g. as arguments of a message, the passive object needs to be deeply copied up until references to active objects. The active objects themselves need not be copied because their semantics can deal with being shared among multiple other objects. We refer to this realization of the containment principle as the *call-by-deep-copy* argument passing.

5.2.3 Active Objects as the Unit of Distribution

We have considered active objects as the unit of concurrency in AmbientTalk. As noted by Briot et al. [BGL98] objects are also a suitable candidate as a unit for distribution. An object typically encapsulates both data and methods together. The consequence of this is that resources allocated and accessed by a method are typically local to the computation, which is a good property in the context of distribution, because it increases the availability of these resources. Another advantage of objects in the context of distribution is that the message passing paradigm to invoke methods on an object aligns well with distributed protocols. What's more, the object reference to which a message is sent abstracts the location of the object. In AmbientTalk, we use active objects both as the unit of concurrency and distribution because the *non-blocking* communication characteristic introduces concurrency as discussed in section 2.4.4. Hence, in the case of AmbientTalk distribution automatically implies concurrency. Since active objects are designed to deal with concurrency, they are a good candidate as the unit of distribution. What is more, the *call-by-deep-copy* parameter passing semantics which enforces the containment principle above can easily be transposed to the context of distribution, because *marshaling*[1] and unmarshaling objects either locally or remotely corresponds to a deep copy; when active objects themselves are marshaled as remote references, similar to the marshaling semantics of network objects defined by Birrell et al. [BNOW93]. Hence, the equivalent parameter passing semantics of a *call-by-deep-copy* emerges in the case of remote messaging between active objects.

AmbientTalk applications are thus conceived as suites of active objects deployed on autonomous devices. Several active objects can run on a device and

[1] In Java this is called serialization of objects, but we use the term *marshaling* to avoid confusion with serialization in the context of concurrency

every active object contains a graph of passive objects. Objects in this graph can refer to active objects that may reside on any device. In other words, AmbientTalk's remote object references are always references to active objects. The rationale of this design is that synchronous messages (as sent to passive objects) cannot be reconciled with the non-blocking communication characteristic presented in section 3.3.

AmbientTalk does not know the concept of proxies on the programming language level. An active object $\texttt{a}_1$ can 'simply' refer to another active object $\texttt{a}_2$ that resides on a different machine. If both machines move out of one another's communication range and the connection is (temporarily) lost, $\texttt{a}_1$ conceptually remains referring to $\texttt{a}_2$ and can keep on sending messages as if nothing went wrong. Such messages are accumulated in $\texttt{a}_1$ and will be transparently delivered after the connection has been re-established. Hence, AmbientTalk's default delivery policy strives for eventual delivery of messages. The mechanism that takes care of this transparency is explained in section 5.4.4. First we discuss both layers of AmbientTalk's object model in technical detail.

5.3 The Passive Object Layer

Remember from section 3.2 that the first requirement for the object model of ambient-oriented programming languages is that it is classless. In a classless model objects are self-sufficient and do not have an implicit sharing relationship with their class. In section 3.2 it was explained that this simplifies the issues that arise in the context of mobile distributed systems.

The object model of AmbientTalk is based on the object model used in Pic% [DDD03]. In this section we discuss this object model.

5.3.1 History and Design Rationale

Pic% is an extension of the language Pico, both languages were designed and conceived by Theo D'Hondt and in a later stage also codesigned by Wolfgang De Meuter. Both these languages are used in an educational and research context. Many of the design principles of Pico were borrowed from the language Scheme [AS96], that is to say that simplicity of the design of the language was a foremost concern. Next to the design principles, Pico also borrowed a number of concepts found in Scheme, such as a dynamic type system, first-class abstract grammar, continuations, closures and an interactive interpreter. Concerning the implementation, a Pico system also features a tail-recursive interpreter and a garbage collector to automatically reclaim memory that is no longer in use. However, Pico also differs from Scheme in a number of profound ways:

- *Syntax:* a common criticism against Scheme is that its prefix syntax is hard to read. For cultural reasons most people seem to read and understand infix expressions better, probably because mathematical expressions are also written using infix notation. Although Pico features infix operators, the goal was still to make its syntax as minimal and regular as possible. We have experienced in our courses and by writing and reading many programs in Pico that giving it a more conventional syntax it enhances the readability.

```
(define (QuickSort V Low High)
  (if (< Low High)
      (letrec ((Left Low)
               (Right High)
               (Pivot
                (vector-ref V
                   (quotient (+ Left Right) 2 )))
               (Save 0))
        (do ((stop #f (> Left Right))) (stop)
          (do () ((>= (vector-ref V Left) Pivot))
            (set! Left (+ Left 1)))
          (do () ((<= (vector-ref V Right) Pivot))
            (set! Right (- Right 1)))
          (if (<= Left Right)
              (begin
                (set! Save (vector-ref V Left))
                (vector-set! V Left
                             (vector-ref V Right))
                (vector-set! V Right Save)
                (set! Left (+ Left 1))
                (set! Right (- Right 1)))))
        (QuickSort V Low Right)
        (QuickSort V Left High))))
```

```
QuickSort(V, Low, High):
 { if(Low < High, {
      Left: Low;
      Right: High;
      Pivot: V[(Left + Right) // 2];
      Save: 0;
      until(Left > Right,
        { while(V[Left] < Pivot,
                Left:= Left+1);
          while(V[Right] > Pivot,
                Right:= Right-1);
          if(Left <= Right,
            { Save:= V[Left];
              V[Left]:= V[Right];
              V[Right]:= Save;
              Left:= Left+1;
              Right:= Right-1 }) });
      QuickSort(V, Low, Right);
      QuickSort(V, Left, High) }) }
```

Table 5.1: QuickSort in Scheme (left) and in Pico (right)

- *Tables:* in Scheme a distinction is made between lists and vectors, while in Pico indexable tables are used to represent both datastructures. Also, in Scheme the abstract grammar is made first-class by using lists, whereas Pico uses tables. I.e. a Scheme program is reified as a nested list, whereas a Pico program is reified as a nested table structure.

- *No Special Forms:* in the implementation of Scheme some special-forms have been introduced, because some language constructs such as control structures need argument evaluation rules that differ from the standard eager evaluation rules. In Pico there is no need for special forms. Instead Pico has richer parameter passing semantics, this is explained in section 5.3.2.

Table 5.1 shows an implementation of the quicksort algorithm in both Scheme and Pico to illustrate the differences in syntax. In Pico variable declarations are made using the "`:`" token as opposed to the `define` special form of Scheme. Similar to Scheme both values and functions are defined using the same notation. Another similarity with Scheme is that functions are first-class values in Pico. In Pico assignments are denoted using "`:=`" as opposed to a number of special forms used in Scheme, such as `set!` and `vector-set!`. Referencing a value in a table is based on square bracket notation, while in Scheme functions such as `vector-ref` and `vector-set!` are used. Blocks of expressions are grouped together in Pico in a `begin` function, similar to the `begin` function in Scheme. However, Pico also provides syntactic sugar based on commonly used curly braces to group expressions. The expressions are separated using semicolons. The final expression in such a group is the return value of the evaluation of the block. The syntax rules of Pico are summarized in table 5.2.

kind of invocation:	name invocations name	table invocations name[e1]	function invocations name(e1, ... ,en)
reference	name	name[e]	name(e1, ... ,en)
definition	name: e	name[e1]: e2	name(e1, ... ,en): e
assignment	name:= e	name[e1]:= e2	name(e1, ... ,en):= e

Table 5.2: Summary of the Pico syntax

5.3.2 Parameter Passing Semantics

Pico has rich parameter binding semantics that is inherited in Pic%. More particularly Pico supports variable sized argument lists and call-by-function arguments. Both types of parameter binding semantics are discussed below.

Variable Sized Argument Lists

As already mentioned above, Pico can be considered as Scheme with a more intuitive canonical syntax. In Scheme the abstract grammar is represented as a nested list of elements and the list of formal parameters is also represented as a list. In fact, in Scheme one can rewrite the anonymous function `(lambda (x y) (+ x y))` as `(lambda r (+ (car r) (cadr r)))`. In the latter expression, the `r` will be bound to the list of arguments used to apply this function. As such, Scheme supports variable sized argument lists in the definition of a function. Also, when calling the function a variable sized actual arguments list can be used. This is realized through the native function `apply`, that takes the function as a first argument and a variable list of actual arguments. For example, `(apply (lambda r (+ (car r) (cadr r))) '(1 2 3))` returns three.

Both variable arguments in the definition and application of functions are also supported in Pico. However, instead of binding the actual arguments to a list they are bound to a table, which is called *call-by-table* parameter passing. Tables are indexable data structures similar to vectors in Scheme. Tables are supported by common syntax based on square brackets, which is summarized in table 5.2. Tables play a prominent role in Pico because all structured data types are based on them. In fact, one could say that tables are to Pico what lists are to Scheme, because the abstract grammar of Pico expressions is represented as tables similar to how lists are used to represent the abstract grammar of Scheme.

In Pico both function definition and application of variable sized arguments have a uniform syntax.

```
f@arg: size(arg)
```

This function definition defines a function `f` with a variable arguments list that will be accumulated in the table `arg`. `size` returns the size of a table. Hence, this function will simply return the number of actual arguments that were supplied upon function invocation and `f(0,1,2)` will return three. The following shows how `begin` was implemented in Pico:

```
begin@args: args[size(args)]
```

Pico's actual argument evaluation rules will evaluate each expression one by one in the order from left to right. Finally, the last expression is returned as the result of this expression according to the semantics of grouped expressions explained above. The curly brace notation to sequence expressions, used in table 5.1 is syntactic sugar for this implementation. As illustrated by the two examples above, curly braces need not be used if the function body only has one expression.

Analogous to Scheme's `apply` it is also possible to invoke a function with a variable sized table using the notation `f@arg`. This expression invokes the function `f` with a table of actual arguments bound to `arg`.

Call-by-Function Parameters

One of the key features that makes Pico an extensible language is its elaborate parameter passing techniques [DDD04]. The variety of parameter passing techniques make that Pico does not need special forms for its control structures. What is more, through the parameter passing techniques of Pico it is possible to implement control structures in the language, such that they do no have to be embedded in the evaluator[2] as special forms. The reason why control structures usually cannot be implemented in the language itself is because all arguments passed to control structures are usually all eagerly evaluated. That is, a function-call is usually evaluated as: 1) search for the appropriate function in the scope; 2) if the function has been found, then evaluate the actual arguments of the function-call one by one and bind them in an extended lexical scope to the formal arguments of the function and 3) apply the body of the function to the extended lexical environment. For example, when `myif` would be implemented as a regular procedure, then using the evaluation-rules of most programming language, the expression `myif(true, 0, 1/0)` would always throw a division by zero exception. In Pico the evaluation rules of the actual arguments of a function-call are determined by the type of formal parameter that is used when the function was declared. This is best illustrated by the example shown below, which implements the *if*-control structure by means of church booleans[3]:

```
{
  true(then, else()):then;
  false(then(), else):else;
  myif(cond, then(), else()):cond(then(), else());

  myif(true, 0, 1/0)
}
```

The function `true` takes two arguments: `then` and `else()`, the former is evaluated eagerly, while the latter is evaluated lazily. `then` is bound in the extended lexical environment to the the evaluation of the actual argument, while `else` is bound as a (first-class) function that has no formal arguments to its *body* that was provided as an actual argument, e.g. `else():{ 1/0 }` for the `myif` function call. The lexical scope of the newly bound function is the scope of the caller of the function. Hence, this form of parameter passing introduces a restricted form of dynamic scope. The former type of argument binding is

[2] In reality, many of the standard control structures are implemented as native functions in the Pico implementation for performance reasons.

[3] In fact, booleans are implemented based on these semantics in the native implementation of Pico.

known as *call-by-value*, the latter type of argument binding has been called *call-by-function*. The evaluation rules of the actual arguments of `false` are the inverse.

Another example where the call-by-function parameter passing rules are used is the function `zero` shown below which calculates the zero of an arbitrary function passed as an argument within the range of `a` and `b` and up to a precision `epsilon`:

```
zero(a, b, f(x), epsilon):{
   c: (a+b)/2;
   if(abs(f(c)) < epsilon,
      c,
      if(f(a)*f(c) < 0,
         zero(a, c, f(x), epsilon),
         zero(c, b, f(x), epsilon))) }
```

The formal parameters `a, b` and `epsilon` are call-by-value parameters. The formal parameter `f` is a call-by-function parameter which takes one argument `x`. This argument `x` is not bound in the scope of the body of the function `zero`. It is in fact the formal parameter of the function `f`. Calling `zero(-1,1,x*2-5,0.001)` thus binds the formal parameter `f` to `f(x):x*2-5` and it is this function `f` which is visible in the body of `zero`.

In languages such as Smalltalk [GR83] and Self [US87] control structures have also been implemented in the language itself using *block structures* instead. The difference between the *call-by-function* and *blocks* is that with *blocks* the expression that is to be lazily evaluated has to be manually embedded in a block structure by the caller. In both Smalltalk and Self square brackets are used for this purpose. Both approaches have advantages and disadvantages: in the call-by-function approach users of a function do not have to think of the special syntax needed for the special evaluation semantics, while in the block approach the semantics of the evaluation rules is more explicit – at the cost of explicit syntax. Nevertheless, when the call-by-function is parameterized, then the caller needs to be aware of the names of the formal parameters, whereas in a block context one can define its own names for the formal parameters.

Another approach that allows the implementation of special forms are macros. However, a macro language is often a different language than the base language such that it becomes less intuitive to use special forms.

Combining Call-by-function and Call-by-table

It is also possible to combine *call-by-table* and *call-by-function*. The resulting semantics of this type of parameter passing is that a function takes a variable sized actual parameter list, the elements of which are all arguments evaluated lazily and bound to a table of functions. This combination of parameter-passing is called *call-by-function-table*. An example of such a function definition is shown below:

```
f@g(x): for(i:1,i<=size(g),i:=i+1,g[i](2))
```

This function definition accepts a variable number of functions of one parameter `x`. It loops over the function-table and evaluates all functions with actual argument `2`. Finally, the result of the evaluation of the last argument will be returned, since it is the last expression evaluated by the for-loop. Hence, calling `f(x, 2*x, 3*x)` yields `6`.

5.3.3 Objects as First-class Dictionaries

In Pic% objects are modeled as *first-class lexical environments* [GJL87, QR96]. Their representation is called a *dictionary*. Lexical environments are similar to those found in Scheme and Pico. Pic% provides richer lexical environments than Pico and Scheme because it makes a distinction between *variables* and *constants*. In the syntax constants are *declared* using "`::`", while variables are *defined* using "`:`". Internally, a dictionary is represented as a pair of linked lists, one containing bindings of variables and the other bindings of constants. Table 5.3 shows the implementation of a generator[4] function for counter objects. The `makeCounter` function declares two methods `decr` and `incr` and returns a new object by calling the native function `capture`, which returns the current dictionary. The formal parameter `n` of `makeCounter` is also in this dictionary as a variable definition, because it is part of the lexical environment when `makeCounter` is invoked.

The dictionary returned by the invocation of `makeCounter(3)` is shown on the right of table 5.3. In this case the variable `counterP` points to that dictionary, which contains two constants `incr` and `decr` and one variable `n`. This dictionary is linked to the *root* dictionary that contains a list of constants, containing the `makeCounter` function and the built-in native functions of Pic% such as `if`, and one variable `counterP`. The dictionary to which the new object is linked determines from which the object inherits and this gives rise to a model of mixin-based inheritance, which is discussed in the next subsection. Nevertheless, we can already state that this *root* object is the topmost object from which all other objects inherit.

In Pic% an attribute is selected using the dot-notation `receiver.attr`. The attribute is looked up only in the constant part of a dictionary, starting from the head of the chain. When the method is not found in the first dictionary, then the constant part of the next dictionary is searched. This process is repeated recursively until a corresponding attribute is found or the `root` dictionary is reached. In the latter case a lookup exception is thrown. An attribute can also be selected using a *receiver-less* expression `attr`, it is then executed in the context of the current receiver. However, the lookup process is different because both the constants and variables are searched. The constants are searched before the variables and in the case no constant or variable matches the next dictionary is searched, recursively until the `root` object has been reached. This difference in attribute lookup between both a receiver-less and receiver-ful expression illustrates how Pic% aligns *definitions* and *declarations* with the visibility rules. The former conform to what is in some language known as *protected* attributes, while the latter conforms to *public* attributes. How late-binding polymorphism is realized in Pic% is discussed below in section 5.3.5. Pic% also has a native method `this()` which returns the current receiver. However, note the difference in lookup semantics between `this().attr` and `attr`. Indeed, the former expression will perform the lookup only in the public (constant) part of the interface, whereas the latter will perform a lookup in the protected (constant and variables) part of the interface. Hence, only the latter can be used to access protected attributes.

[4] Such a generator is actually a mixin method, further discussed in section 5.3.4

```
{
  makeCounter(n)::{
    decr()::n:=n-1;
    incr()::n:=n+1;
    get()::n;
    capture()
  };
  counterP: makeCounter(3)
}
```

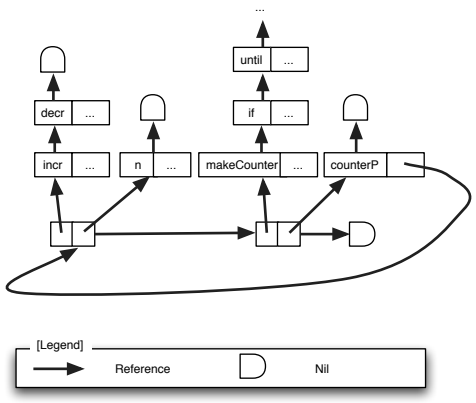

Table 5.3: Example Counter object in Pic% (left) and the resulting environment (right)

5.3.4 Mixin-Based Inheritance

In a prototype-based language (PBL) sharing is achieved through *delegation* [Lie86], sometimes called *object-based inheritance*. Delegation allows objects to flexibly share data and code based on the sender of the message whereas in a class-based language (CBL) there is only one sharing strategy: code and data is shared between objects of the same class. Much of the dynamic nature of PBLs stems from the fact that delegation-based PBLs allow one to change the inheritance pointer of an object at run-time, whereas inheritance links in CBLs are usually immutable at runtime.

Usually, the subclass relationship influences the visibility rules of the object's interface. For example, the `protected` keyword found in most mainstream CBLs such as C++, Java and C# give the privilege to subclasses to access certain implementation details that have been annotated with this keyword, while these implementation details are not visible for regular method invocations. Other languages, such as Smalltalk, grant this privilege to subclasses by default. Such special privileges allow subclasses to break the encapsulation of their parents as was noted by Snyder [Sny86]. However, the consequences of this observation are confined, because the subclass relationship is static and fixed at compile-time and as a result the encapsulation can only be broken by a well-defined set of classes. Unfortunately, if we translate the consequences of the encapsulation problems in the perspective of PBLs with delegation-based inheritance then the repercussions are worse because the delegation-relationship is dynamic and as a result it becomes impossible to delimit the places where encapsulation has been broken. The encapsulation problem is not only important from a software engineering point of view, but it also has consequences from a security point of view [DTM+05]. Indeed, when objects can break the encapsulation they can get access to privileged information. Consider an object which is responsible for paying the bill and therefore holds a reference to a credit card number. Obviously, if one can break the encapsulation of that object the credit card

```
makeContainer()::{
  makeVector()::{
    v: [];
    siz: 0;
    addFront(el)::{ ... };
    addBack(el)::{ ... };
    removeFront(el)::{ ... };
    removeBack(el)::{ ... };
    isEmpty()::{ ... };
    isFull()::{ ... };
    contains(el)::{ ... };
    iterate(code(el))::{ ... };
    map(transformer(el)::{ ... };
    remove(index)::{ ... };
    get(index)::{ ... };
    capture()
  };
  makeList()::{
    lst: [];
    siz: 0;
    addFront(el)::{ ... };
    addBack(el)::{ ... };
    removeFront(el)::{ ... };
    removeBack(el)::{ ... };
    isEmpty()::{ ... };
    isFull()::{ ... };
    contains(el)::{ ... };
    iterate(code(el))::{ ... };
    map(transformer(el)::{ ... };
    capture()
  };
  makeQueue()::{
    pop()::this().removeFront();
    push(el)::this().addBack(el);
    capture()
  };
  makeStack()::{
    pop()::this().removeFront();
    push(el)::this().addFront(el);
    capture()
  };
  capture()
};

containerP: root.makeContainer();
vectorP: containerP.makeVector();
listP: containerP.makeList();
stackOnVectorP: vectorP.makeStack();
stackOnListP: listP.makeStack()
```

Table 5.4: Mixins used to Structure Collections Hierarchy

number can be used by any other object.

To address this problem Steyaert and De Meuter proposed [SM95] an object-based variation of mixins [BC90]. In the approach, child-objects are created by calling a *mixin* method on the parent object. Upon invocation of a mixin method on an object, an extension of that object, with the behavior defined in the mixin method, is returned. Mixin methods delimit the encapsulation problem, because it is the object that declares the mixin method who determines its children and therefore prescribes what objects have access to its implementation details.

Table 5.4 shows an example where mixins are applied to a small collection hierarchy. In the example a `makeContainer` method declared in the `root` object which returns an object containing four *mixin* methods `makeVector`, `makeList`, `makeQueue` and `makeStack`. Both `makeVector` and `makeList` implement a uniform interface to access the contents of the container with the exception that a *vector* implements two extra methods `remove` and `get` to randomly access its contents. The two mixin methods `makeQueue` and `makeStack` have the same interface, but with a different implementation. Using nested mixin methods we can now create multiple linearized combinations of object hierarchies. The order of the object inheritance hierarchy depends on the order in which the mixin methods have been invoked. For example, figure 5.1 shows the resulting object-tree from the evaluation of `containerP.makeList().makeStack()` and `containerP.makeStack().makeList()`. Both expressions form hierarchies composed of the same objects but the *stack* and *list* objects are linked in a different order. Also note that it is possible to reuse the queue and stack implementations on both list and vector objects, similar to the reuse of mixin classes as proposed by Bracha and Cook [BC90].

A drawback of the mixin-based approach for inheritance is that the definition of child objects is nested in the definition of parent objects. As a consequence child objects cannot be defined *outside* the parent object, which we refer to as

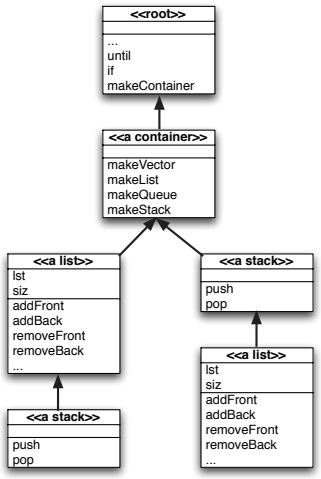

Figure 5.1: Resulting object-tree from the evaluation of `containerP.makeList().makeStack()` (left) and `containerP.makeStack().makeList()` (right)

```
protectedCounter: counterP.extend({
  limit: 3;
  incr()::{
    if(n=limit,error("overflow"), .incr()) };
  decr()::{
    if(n= -limit,error("underflow"), .decr()) };
  capture()
});
```

Table 5.5: Example: extension from the outside - a protected counter

extension from the outside. For this reason the `root` object contains a native method `extend` which takes an object definition as its argument and extends the receiver with the object definition. An example of an extension from the outside on the counter prototype from table 5.3 is shown in table 5.5. The extension overrides two methods `incr` and `decr` in which the upper- and lowerbound is checked before delegating to the parent object via a super-send.

In Pic% the syntax of a supercall is a dot followed by the method invocation, for example `.incr()` is the super-call for the `incr` method. Pic% also features a native `super` function which returns the parent of the object in which the current evaluation takes place. However, `super().incr()` has a different semantics than `.incr()`. Using the former syntax the *this* will be rebound to the parent object because it's a normal message send, while using the latter syntax *this* will not be rebound and refer to the object that received the method invocation. A more detailed discussion on the late-binding polymorphism of Pic% follows in the next subsection.

A limitation of extensions from the outside is that they are not reusable. In-

```
a)
stealCreditCardNumber(aPaymentObject)::{
  aPaymentObject.extend({
    getCreditCardNumber()::{
      creditCardNumber };
    capture()
  }).getCreditCardNumber()
}
```

```
b)
makePaymentObject(creditCardNumber)::{
  pay(aTransaction)::{ ... };
  extend(anExtension)::
    error("cannot extend this object");
  capture()
}
```

Table 5.6: Example: (a) stealing the credit card number from a payment object and (b) prevent this by overriding the `extend` method in the payment object

deed, a mixin method can be applied to all objects that understand the method, while an extension from the outside is applied to one object and cannot immediately be applied to other objects. However, reusability can be obtained by creating a method which takes one argument, the object to be extended and invoke the `extend` method on the receiver with the extension code as its argument. Another, perhaps more severe, drawback is that this mechanism opens up the internals of the object to any other object. Indeed, any object can make an extension from an arbitrary object so that the extension opens up the implementation details of the object as noted by Steyaert and De Meuter [SM95]. An example of this is shown by table 5.6a. However, because `extend` is a method it can be overridden to throw an error, thereby preventing other objects to extend from the outside. This is illustrated by the example shown in table 5.6b.

5.3.5 On Late-Binding Polymorphism and First-Class Methods

As briefly mentioned in the introduction, Pico features first-class closures, similar to Scheme. First-class closures are sometimes also called first-class functions, but a closure is more than a pointer to the function definition. A closure is a pair with a pointer to the function definition *and* a pointer to the lexical scope which captures the undefined variables in the function definition. Without the lexical scope, first-class functions would be dynamically scoped. Hence, in Pico and Scheme closures are essential to introduce correct semantics for their first-class functions. In section 4.3.1 we have explained that first-class closures formed an important element to emulate an object system for the actor model. Likewise, closures also play an important role in Scheme and Pico to emulate an object system. However, in Pic% closures are not used to emulate objects through a dispatch function, since dictionaries are used to represent objects. Nevertheless, in Pic% (first-class) closures play an important role in the semantics of the object model, but there are some subtile differences between closures in Pic% and closures in other languages [DD03]. In Pico and Scheme, closures are created at function definition-time, whereas they are created at method invocation time in Pic%. As a result, the environment of Pic% objects contains *functions* instead of *closures*. This is illustrated by figure 5.2, which shows an environment of a counter object in Pic% (created with the code shown in table 5.3) and in Scheme (created with the code shown in table 5.7). The former environment contains a direct reference to the functions, whereas the latter environment contains a reference to the closures. This is shown in the figure for the `decr` method in

```
(define (makeCounter n)
  (define (decr)
    (set! n (- n 1)))
  (define (incr)
    (set! n (+ n 1)))
  (define (get) n)
  (define (dispatch m)
    (cond ((eq? m 'incr) incr)
          ((eq? m 'decr) decr)
          ((eq? m 'get) get)
          (else (display "Error - unknown message."))))
  dispatch)
```

Table 5.7: Object Generator function returning Counter Objects in Scheme

both languages.

In Pic% closures are created at invocation-time. Thus when a function is selected from a dictionary through an invocation, that function is wrapped in a closure and the lexical scope will refer to the dictionary from which the method was originally selected. For example, the expression `counterP.get` will return a closure with the lexical scope pointer directed at the `counterP` dictionary and the function pointer directed at the `get` function. Note that the expression `(counterP 'get)` in Scheme would return an equivalent closure. However, in the face of object-based inheritance the closure semantics in Scheme would no longer provide the semantics of late-binding. Indeed, in Scheme the lexical pointer of the closure would always point to environment in which it was embedded, while in Pic% the lexical pointer will point to the original receiver of the invocation. For example, considering the `protectedCounter` extension from table 5.5, the evaluation of `protectedCounter.get` will result in a closure the lexical pointer of which refers to the dictionary `protectedCounter` while its function pointer will refer to the `get` function found in the dictionary of `counterP`. Hence, when the closure is applied to its arguments it will be executed in the context of the `protectedCounter` dictionary and therefore have late-binding semantics.

The scheme of the closures that are created at invocation-time has three advantages. First, it enables late-binding semantics for object-based inheritance built by the mixin methods as explained above. Secondly, the methods are reentrant across object-hierarchies, because the dictionary (containing methods) does not contain references to a closure, but to functions. Hence, in the case of inheritance the same function will be always wrapped in a closure referring to the dynamic receiver of the message. As a result the function is reentrant and does not have to be duplicated when it is reused in the context of inheritance. Finally, the closures can be regarded as first-class methods, because closures can be passed around and are executed in the context of the object from which they were selected [DD03]. Below we show that the object model of Pic% also supports reentrancy of methods in the context of object copies.

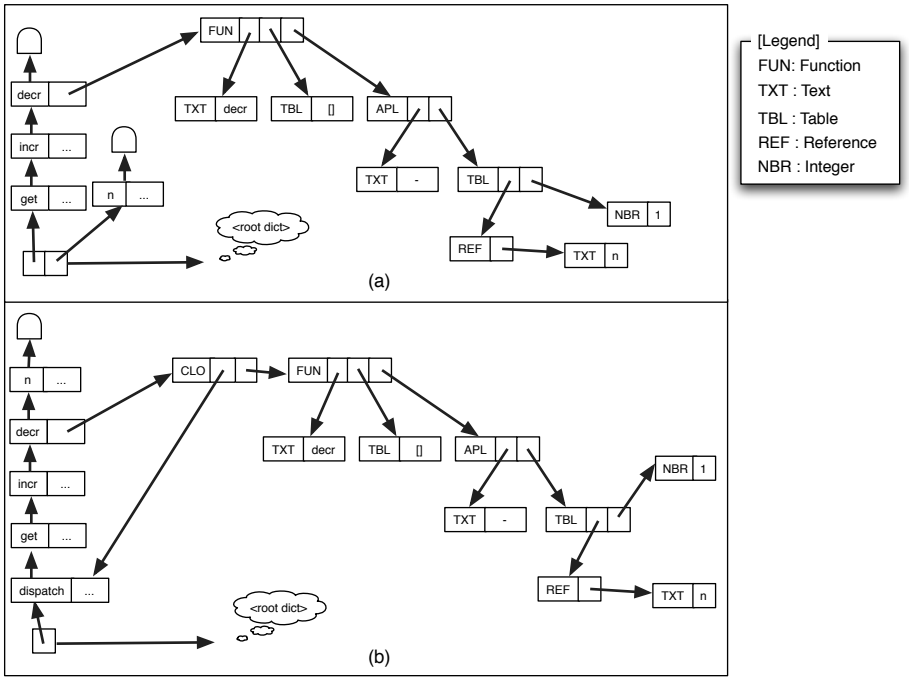

Figure 5.2: Differences in the Environments between (a) Pic% and (b) Scheme

5.3.6 Cloning Objects

A characteristic shared by all PBLs is that they support cloning objects. Indeed, the prototype-based paradigm is based on creating a prototypical object for a concept in the solution domain of an application. That prototype then serves to create other similar objects by copying the prototype and changing that copy to fit the needs of the application. There are two concerns when copying an object. First, one needs to specify how much of the object graph should be copied along. Second, after the clone is created it often needs to be adapted and properly initialized.

In Pic% the first concern is addressed by means of a universal semantics for cloning. When an object is cloned then the variable bindings are deep copied, while the constants are shallow copied. Figure 5.3 shows the memory layout for `counterP` and a clone of this object. This universal semantics together with the closure semantics, explained above, renders public methods reentrant entities, because they are declared as constants.

Two native cloning methods `clone` and `copy`, which are available in the *root* object, have been provided based on this semantics. The native method `clone` has been introduced in the language and allows one to express how much of the delegation hierarchy is to be copied when an object is cloned. `clone` takes a reference to an object as its argument and the cloning operation starts from the object to be copied, walks through each ancestor of the object and matches the argument against that ancestor, starting with the direct parent. If they do not match then that ancestor is cloned along according to the prin-

ciples explained in the previous paragraph. The cloning operation stops when the argument matches with an ancestor and the matching ancestor becomes the parent of the current object that has been cloned. Hence, the mechanism allows one to clone objects up until a specific ancestor. That ancestor is then shared by the object that was cloned and the clone itself. For example, invoking the method `clone(root)` on `protectedCounter` (whose code was shown in table 5.5 on page 107) creates a clone from `protectedCounter` that shares the *root* object with its clone. `protectedCounter.clone(counterP)` would create another clone of the `protectedCounter` that shares the counter prototype. Hence, in the latter case both copies of the clone would share the variable `n` found in `counterP` and operations on `protectedCounter` and its clone would be visible for one another. Another native method `copy` takes an initialization expression as its argument. It creates a clone of its receiver and the initialization expression is then executed in the context of the clone, enabling the clone to be properly initialized. For example, `protectedCounter.copy(n:=2)` creates a clone of the protected counter object and initializes its state variable `n` to two. Cloning an object is initiated by method invocation, similar to the `extend` method, such that the encapsulation of objects can also be preserved by overriding the `copy` method.

Both cloning methods could be unified in a single native method, which takes two arguments, the ancestor and an initialization expression. However, the current implementation of AmbientTalk does not contain such native.

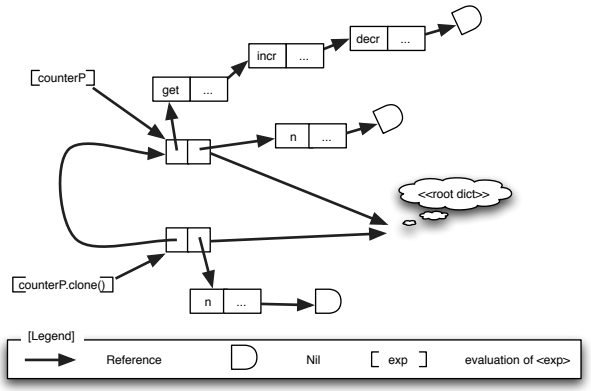

Figure 5.3: Memory Layout of Counter Object and its Clone

5.3.7 Summary

Pic% extends Pico with prototype-based object-oriented concepts. The syntax extensions to Pico are summarized in table 5.8. We have shown that in the Pic% object system, on which AmbientTalk has been built, supports three mechanisms for creating new objects. First, there are mixin methods that can be used to dynamically construct object hierarchies at run-time without forfeiting the encapsulation of other objects. Hence, mixin methods reconcile object encapsulation with the dynamic object hierarchies that are typically found in PBLs that support delegation as a mechanism for sharing data and methods.

kind of invocation:	name invocations name	table invocations name[e1]	method invocations name(e1, ... ,en)
definition	name: e	name[e1]: e2	name(e1, ... ,en): e
assignment	name:= e	name[e1]:= e2	name(e1, ... ,en):= e
receiver-less	inv	inv[e1, ... ,en]	inv(e1, ... ,en)
declaration	name:: e	name[e1]:: e2	name(e1, ... ,en):: e
super sends	.name	.name[e1, ...]	.name(e1, ...)
qualification	e.inv	e.inv[e1, ...]	e.inv(e1, ...)

Table 5.8: Summary of the Pic% syntax

Secondly, we have extension from the outside that allows objects to be extended at run-time with custom behavior. However, extension from the outside has the drawback that it breaks the encapsulation of an object. It is therefore possible to override this method such that the encapsulation of the object can be preserved. A third mechanism for creating new objects is to clone an existing object. We have explained two such cloning mechanisms.

5.4 The Active Object Layer

The active object layer of AmbientTalk is based on the ambient actor model we introduced in the previous chapter. We have seen that the ambient actor model was missing a true object system, it had to be emulated through the use of closures. Another conclusion of the ambient actor model was that, although the extensions fulfill the ambient-oriented paradigm, it could be regarded as a "low level language" for ambient-oriented programming languages. In this section we will integrate the passive object layer of Pic%, explained in the previous section, with the ambient actor model to resolve the former observation. The latter one is addressed in the following chapters.

5.4.1 Active Objects as Actors

In AmbientTalk, active objects are modeled following the principles of the actor model. Throughout the remainder of this text we will use the terms *active object* and *actors* on the one hand and the terms *passive object* and *object* on the other hand, interchangeably in the context of AmbientTalk. An active object conceptually consists of its behavior, which is implemented by a passive object, a set of mailboxes and a thread of execution. Figure 5.4 shows a conceptual drawing of two active objects containing a graph of passive objects obeying the containment principle explained above. An actor is created using the `actor` method, which is part of the interface of the *root* object. This method takes a passive object as its argument and returns the *mail address* of a new active object with a deep copy of that passive object as its behavior. A deep copy of that object is taken to comply to the *containment principle* introduced above, otherwise a passive object is shared between the created active object and its creator. Since both active objects conceptually have a thread of execution, they could otherwise potentially update shared state, with race conditions as a consequence.

Table 5.9 shows two implementations of the behavior of a counter active object. The implementation on the left creates the passive behavior of the counter object by creating an extension from the outside of the *root* object. The object has one variable `n`, for representing the value of the counter. Besides the state, a number of methods have been implemented by the object: `new` returns a modified clone of the counter object; `increment` and `decrement` update the state of the object; `get` method sends the value of the counter to `customer` and finally the `init` method initializes the passive object in its context as an actor. Note that the `new` method initializes the passive object, whereas the `init` initializes the active object. As a convention, the name of passive objects that are used for the behavior of active objects usually ends in `Behavior`. After the definition of the actor's behavior the actor is created using the `actor` method. The actor is then sent an `increment` and `decrement` message. Finally, the state of the counter is retrieved by the sending the callback message `get`. Its argument `thisActor()` denotes the current actor's mail address. The implementation in the right hand side column of table 5.9 uses the `become` method to change the behavior of the counter. This method takes a new behavior as its argument and does not return a result. Unlike the `actor` method, this method does not take a deep copy of the object that was passed as its argument. This is not necessary, because the object does not change from the context of one active object to another. Obeying the semantics of the *become* operation in the ambient actor model, introduced in the previous chapter, this method does not introduce parallel activity. However, unlike the ambient actor model the `become` method does not interrupt the control flow. In other words, the expressions following the `become` method will be executed. It might seem that the `become` method is redundant, because we can update the state of an active object using the assignment operator. However, the *become* has several uses:

- First, it complements the assignment in that it allows one to write in a functional style. Such a use of the `become` can be used to easily implement the state pattern [GHJV94].

- Another use for the `become` method, that is not expressible by assignments, is changing the *interface* of an active object. Changing an interface of an active object may seem odd at first, but it is useful in the context of adaptability to new environments. As mobile devices roam from one network to another they might need to interact with software components providing the same services, but that provide a different interface to interact with and might also require a different interface.

- Finally, the `become` can also be used to express behavioral *synchronization*. Namely, when a message is received by an actor for which the behavior of that actor does not define an implementation then that message is stored. Hence, when an actor changes its behavior and its replaced behavior contains an implementation then the actor can process that message. Using this simple mode of delaying messages until a behavior for consuming these messages is found can be used for synchronizing the interactions between actors. This is a first very rudimentary synchronization mechanism, but in section 7.2 we will introduce more advanced synchronization schemes.

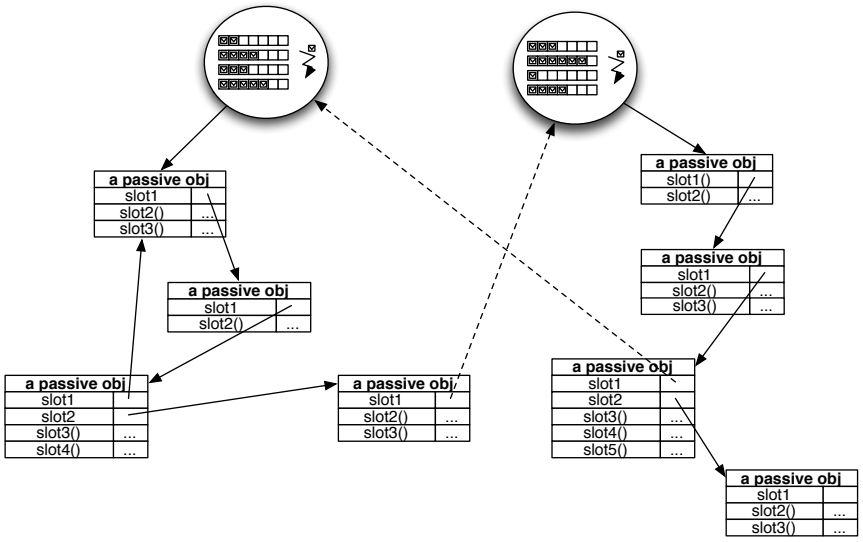

Figure 5.4: Active Objects Conceptual Model. Two active objects containing a graph of passive objects. None of the passive objects are shared, but the each active objects shares a references to the other.

5.4.2 Message Passing Semantics

In AmbientTalk message passing syntax is different from the method invocation syntax introduced above. The code shown in table 5.9 uses the `#`-notation for sending a message to an actor. Although they apparently both seem to do the same thing, that is requesting an object to perform a certain task, there are good reasons to distinguish between them:

- First, a message is sent asynchronously, whereas a method invocation is handled synchronously. In the latter case regular method invocation semantics applies, while in the former case no value is returned. Hence, in the default behavior of AmbientTalk one has to utilize callback methods to process results of asynchronous message sends. In chapter 7 we will extend the kernel to introduce the notion of futures [Hal85, LS88] to represent the results of an asynchronous invocation. Nevertheless, these futures are not fully transparent and need to be handled explicitly.

- Second, the arguments that are passed as parameters of an asynchronous invocation will potentially cross the context of one active object to another. Hence, to preserve the *containment principle* these messages and the associated arguments are passed via a *call-by-deep-copy* as opposed to *call-by-value* in the case of a method invocation.

- Another, third reason is that asynchronous invocations not only cross the border of active objects, but may also cross the border of a device. Therefore, at these places in the code one needs to be aware of the consequences of remote communication, such as failed communication.

```
counterBehavior: root.extend({
  n: 0;
  new(aNumber)::{ copy(n:=aNumber) };
  increment()::{ n:=n+1 };
  decrement()::{ n:=n-1 };
  get(customer):: { customer#result(n) };
  init()::{ display("initialized actor") }
});

mycounter: actor(counterBehavior.new(5));
mycounter#increment();
mycounter#decrement();
mycounter#get(thisActor())
```

```
counterBehavior: root.extend({
  n: 0;
  new(aNumber)::{ copy(n:=aNumber) };
  increment()::{
    become(counter.new(n+1)) };
  decrement()::{
    become(counter.new(n-1)) };
  get(customer):: { customer#result(n) };
  init()::{ display("initialized actor") }
});

mycounter: actor(counterBehavior.new(5));
mycounter#increment();
mycounter#decrement();
mycounter#get(thisActor())
```

Table 5.9: Implementation of a counter actor using updateable state (left) and using the become operation (right)

The arguments above illustrate why it is a good idea to differentiate in syntax between both modes of communication. This is in contrast to other middleware and programming languages, such as ProActive [ACG00] and ChitChat [De 04], where the difference between the two types of communication has been made oblivious. The arguments also amount to the fact that keeping the distribution transparent for the programmer is in many cases impossible in the context of mobile networks.

A difference between message passing and method invocation we have discussed above is that actual parameters are *call-by-deep-copy* as opposed to *call-by-value*. However, method invocation in Pic% features another type of parameter passing, namely *call-by-function* as discussed in section 5.3.2. It depends on the definition of the method's formal parameters if its arguments are bound as *call-by-value* or *call-by-function*. Transposing *call-by-function* in the context of message passing has a number of consequences. First, the manner in which the arguments are bound and eagerly or lazily evaluated depends on the method definition and thus on the side of the callee, while the actual arguments are evaluated at the side of the caller. The actual arguments are either evaluated immediately or lazily depending on whether the formal argument prescribes call-by-value or call-by-function. However, in the case of message passing the caller and callee do not necessarily reside on the same device and if at message send-time, the recipient actor is not available then the active object would have to postpone the call until it can access the method definition on the side of the callee. However, postponing a call would imply blocking communication and therefore violate the *non-blocking* communication characteristic we advocated in section 3.3. Nevertheless, this problem could be resolved by embedding the interface definitions of an active object in the remote references so that the active object at the side of the caller could determine the evaluation semantics of the arguments. However, such a choice would also imply that introducing the `become` operation is impossible because this operation potentially changes the interface of the active object at run-time. Even if one would make this tradeoff, not all problems are solved. Namely, the function that is created as a result of *call-by-function* parameters has its lexical environment bound to the side of the

kind of invocation:	name invocations name	table invocations name[e1]	method invocations name(e1, ... ,en)
definition	name: e	name[e1]: e2	name(e1, ... ,en): e
assignment	name:= e	name[e1]:= e2	name(e1, ... ,en):= e
receiver-less	inv	inv[e1, ... ,en]	inv(e1, ... ,en)
declaration	name:: e	name[e1]:: e2	name(e1, ... ,en):: e
super sends	.name	.name[e1, ...]	.name(e1, ...)
qualification	e.inv	e.inv[e1, ...]	e.inv(e1, ...)
asynchronous	e#inv	-	e#inv(e1, ...)

Table 5.10: Summary of AmbientTalk syntax

caller. Hence, if the callee invokes the function then it is executed in its lexical context of its caller. As a result, when the caller and callee reside on different machines and they cannot communicate such a call would have to block until the caller is available so that the lexical environment can be accessed. Again, this would imply violating the *non-blocking* communication characteristic. Alternatively, a copy of the lexical environment could be passed along so that no such availability problems arise. However, this would imply that the internal state of the caller is transferred to the callee posing security and performance issues. Moreover, changes that occur in the context of invoking the *call-by-function* parameters would imply that the copy of the lexical environment is changed and not the lexical environment of the caller. As a result the *call-by-function* parameter passing would have a different semantics for message passing than for method invocation. For all these reasons, message passing semantics does not support *call-by-function* parameters and *call-by-function* is restricted to synchronous method invocation.

In section 5.3.5 we discussed the representation and syntax for first-class methods. A method can be reified by merely selecting it from an object through the dot-operator. In a similar vein syntax is provided to reify messages. A message is reified by selecting it through the #-operator. Hence, `anActor#m` will reify the message `m` with `anActor` as its target. Table 5.10 completes the syntax summary of AmbientTalk.

5.4.3 First-Class Messages

Messages are represented as passive objects that are clones from the `message` prototype, shown in table 5.11. This prototype object contains methods to manipulate the contents of a message. A message contains the following attributes:

- `source` refers to the actor that created the message
- `target` is the recipient of the message
- `name` is the name of the method that is to be invoked when this message is processed
- `argList` refers to the list of arguments used to invoke the associated method

```
message::root.extend({
  source : void;
  target : void;
  name   : void;
  argList: void;

  new(aSource, aTarget, aName, anArgList)::copy({
    source:=aSource;
    target:=aTarget;
    name:=aName;
    argList:=anArgList;
  });

  getSource()::source;
  getTarget()::target;
  getName()::name;
  getArgs()::argList;

  setSource(aSource)::source:=aSource;
  setTarget(aTarget)::target:=aTarget;
  setName(aName)::name:=aName;
  setArgs(anArgList)::argList:=anArgList;
});
```

Table 5.11: Message Prototype Object

Next to these attributes the message contains methods to read and write these attributes. The absorption and reification of the first-class messages in the AmbientTalk interpreter is further detailed in section 6.5.1.

5.4.4 First-Class Mailboxes

Eight native first-class mailboxes were introduced in the ambient actor model for the reification of the communication traces and the reification of the environmental context discussed in sections 3.4 and 3.5, respectively. The different mailboxes from the ambient actor model are directly adopted in the AmbientTalk kernel. These mailboxes are built into the kernel through the native methods `__add`, `__delete` and `__messages` to change and retrieve the contents of mailboxes, much in the spirit of the operations to manipulate mailboxes in the ambient actor model. Based on these native methods, a mailbox has been represented as a passive object, shown in table 5.12. Each of the native mailboxes are bound by default in an active object using the names `inbox`, `outbox`, `sentbox`, `rcvbox`, `providedbox`, `requiredbox`, `joinbox` and `disjoinbox`.

AmbientTalk introduces the concept of *mailbox observers*. An observer monitors changes made to a mailbox, more particularly adding and deleting contents to a mailbox. Observers are registered to a mailbox using the method methods `addAddObserver` and `addDeleteObserver`. The former method registers an observer that is notified when contents is added to the mailbox, whereas the latter method registers an observer that is notified when contents is deleted from a mailbox.

Mailbox observers enable active objects to monitor changes in the communication process. An alternative to monitor changes in the communication process would be to open up the implementation of the mailbox natives and the underlying low-level communication process, such as in CoDa [McA95]. However, this

```
mailbox::root.extend({
  name: "in";

  new(aName)::copy({ name:=aName });

  add(el)::__add(name, el);
  get(nr)::__messages(name)[nr];
  delete(el)::__delete(name, el);
  length()::size(__messages(name));

  asVector()::{
    tbl: messages(name);
    vector.newWithTable(tbl)
  };

  iterate(it(el))::
    { t: it; this().asVector().iterate(t(el)) };
  ...
});
```

Table 5.12: Mailbox Prototype Object

alternative could allow one to design an alternative communication process that no longer supports the AmOP criteria, i.e. introduce blocking communication primitives in AmbientTalk. In contrast, the use of mailbox observers restrict changes to the low-level communication protocol, but still allow one to monitor and change the high-level communication protocols between actors.

Observers are modeled after the observer design pattern [GHJV94]. However, there are two differences as to how the observer pattern is implemented:

- First, active objects are observers as opposed to passive objects. Active objects are notified by means of asynchronous method invocations rather than synchronous method invocations. As explained in section 4.5.5, there are two different processes that manipulate mailboxes, the actor system and the active object that owns the mailbox. Note that the method activation mode (synchronous method invocation vs. message passing) for observer notifications determines the process that will execute the observer code: if the observer is notified using a synchronous method invocation, then the code would sometimes be executed by the communication scheduler. For example, if the actor system receives a message and places it in the `inbox`, then the actor system would synchronously notify such that the observers would be executed by the process of the actor system. On the other hand, in the case the message passing mechanism is used to invoke the observer it is the active object that will process the observation.

  Hence, the choice of the process that will execute the observer depends on the method activation of an observer. Using the process of an active object seems to be the most intuitive, because it is an active object that registers its observers onto its mailboxes. What is more, the registered observers may perform operations that are identified with an active object such as sending messages or changing its state. Hence, in the case of synchronous method activations two different processes (the active object and the communication scheduler) could concurrently manipulate the same state such that race conditions can occur.

- A second difference with the standard observer pattern is that instead of registering observer *objects*, first-class *messages* are registered. When a mailbox is changed and a first-class message is associated with that event then that first-class message is sent. The recipient of that message determines the active object that acts as the observer. Subscribing messages instead of active objects increases the flexibility of the observer pattern, because the interface of the observer object does not need to be fixed with respect to the protocol of the design pattern. First-class message that are registered as mailbox observers are henceforth called *observer messages*.

Observing mailboxes in an asynchronous fashion involves a special scheduling policy of the observation messages to ensure that they are processed before all other messages. Processing the observer messages before all other messages has the advantage that an object can respond to the event in a timely fashion. For example, suppose an observer watches for additions to the `inbox` of an active object. If a message *msg* is placed in the `inbox` then the observer message is scheduled before all other messages and thus also the message *msg*. This system is more expressive because it enables intervention before the message *msg* is processed. For example, if the action associated with this mailbox observer would be to remove the message *msg* from the `inbox` then *msg* will never be processed. If the mailbox observer would not be scheduled before other messages then it would be processed after *msg* has already been processed such that removing it from `inbox` would be impossible. For this reason mailbox observers are scheduled with priority. Scheduling with priority entails that observer messages are placed in the `inbox` before all other messages such that they are processed before all other messages. Hence, if a mailbox is changed then observer messages associated to this change will be the first messages processed after the current message has finished executing.

5.4.5 Example: Friend Finder Application

To put everything we have explained above together and to illustrate the use of first-class mailboxes and mailbox observers we discuss the implementation of a friend finder application in the AmbientTalk kernel. Several such applications have already been developed for cellular phones equipped with a bluetooth interface. The purpose of this application is to help people with similar interests find one another when they are in each others vicinity. For example, two people with matching interests sitting in the same railroad carriage would have their mobile phones notifying them by playing a tune. Both FriendFinder applications could use different matching criteria. For example, one may have more stringent criteria and require two matching hobbies instead of one. In that case only one device would make a sound and one would not be able to find the other. Hence, both mobile phones need to play a tune.

A proof of concept implementation of an active object epitomizing such an application can be found in table 5.13. The `friendFinderBehavior` object has two variables `myAge` and `myHobbies` representing the age and the list of interests of the owner. Furthermore, there is a constant `ageRange` that defines the range of age that will match as a friend. The most interesting methods are `match`, which checks the matching criteria and notifies the owner when in the case of a match, and the `init` method which adds the <`friendfinder`> pattern

```
friendFinderBehavior :: root.extend({
  myAge      : 18;
  myHobbies  : vector.new();
  ageRange   :: 2;

  new(anAge)::copy({
    myAge:=anAge;
    myHobbies:=vector.new() });

  match(age, hobbies)::{
    if(and(abs(age - myAge) <= ageRange,
           hasMatchingHobbies(hobbies)),
       friendFinderUI#notify()) };

  addHobby(aHobby)::{
    myHobbies.add(aHobby) };

  hasMatchingHobbies(hobbies)::{
    myHobbies.detect(hobbies.contains(el))
  };

  init()::{
    providedbox.add("<friendfinder>");
    requiredbox.add("<friendfinder>");
    joinbox.addAddObserver(
      thisActor()#onJoined);
    disjoinbox.addAddObserver(
      thisActor()#onDisjoined)
  };

  onJoined(aResolution)::{
    ff: provider(aResolution);
    ff#match(myAge, myHobbies)
  };

  onDisjoined(aResolution)::{
    toDelete: outbox.asVector().select({
     dissolvedFF: provider(aResolution);
     el.getTarget() = dissolvedFF });
    toDelete.iterate({ outbox.delete(el) });
    ... other mailboxes emptied too ...
  }
}));

makeFriendFinder(age)::
  actor(friendFinderBehavior.new(age));
```

Table 5.13: Implementation of a FriendFinder

both to the `requiredbox` and `providedbox` mailbox. This pattern is distinct from other applications such that friend finder applications can detect one another in the ambient. `init` also subscribes two mailbox observers `onJoined` and `onDisjoined` on the `joinbox` and `disjoinbox` respectively. Hence, two `FriendFinder` objects that are running on different mobile devices and that are in the communication range of one another will have their `joinbox` updated. Similarly, when the two devices are moved out of the communication range of one another their `disjoinbox` will be updated. This is according to the operational semantics defined and discussed in the previous chapter. The mailbox observer `onJoined`, which is received when the `joinbox` is updated will send a `match` message with the information of the owner to the `FriendFinder` object in the communication range. The `FriendFinder` object running on the other device will do the same. In the case of a match the friend finder will notify the user interface, which will in turn start playing a tune. The mailbox observer `onDisjoined` cleans up communication traces of mobile devices that disappeared. Cleaning up the communication traces is necessary, because in this particular application chances are slim that a message that is still in the `outbox` will be delivered at a later time.

5.5 Conclusion

In this chapter we have discussed how the AmbientTalk programming language realizes the operational semantics of the ambient actor model. Furthermore, we have also elaborated on the design decisions of both the sequential and distributed object model. The distributed object model of AmbientTalk extends Pic%'s sequential object model by introducing active objects and asynchronous

invocations.

We illustrated the use of AmbientTalk's object model by means of a number of examples. In retrospect, if we look at these examples then we can conclude that although they are expressed in a more high-level object model, fundamentally they still have the same problems that we discussed for examples based on the operational semantics in the previous chapter. In other words, although AmbientTalk its object model is more expressive it still lacks the necessary expressiveness with regard to the concurrent and distributed aspects we typically encounter in regular distributed programs. This is for example illustrated by the need for callbacks to process results of requests sent to active objects.

Another open problem at this point in the dissertation is that although the semantics of the ambient actor model were well defined as an extension of the actor model in the previous chapter, the semantics of AmbientTalk were only described informally. In the next chapter we address both issues.

Chapter 6

AmbientTalk and Metalinguistic Abstraction

6.1 Introduction

A *metacircular* evaluator is an evaluator written in the same language it evaluates. The study of a metacircular evaluator has several advantages. First, a metacircular interpreter gives the precise semantics of the language. Secondly, a metacircular interpreter is a good experimentation platform for studying and experimenting with new language features, because a metacircular interpreter is often less complex than its native counterpart. The fact that a metacircular implementation is often less complex than its native counterpart takes us to a third advantage: it is often easier to study the semantics, because it is less complex, yet it still holds the essence. Thirdly, the essence of meta circularity is that one has to understand precisely one program written in a programming language in order to understand all programs written in that language because that program defines the semantics of the language. However, there are also some practical disadvantages. First, a metacircular interpreter cannot bootstrap the interpreter. One needs a native implementation of the interpreter before the metacircular one can be used. Second, a metacircular interpreter is often a lot less efficient that its native counterpart.

The goal of this chapter is twofold. First, we want to establish the exact semantics of the kernel language AmbientTalk. We have discussed the overall design decisions and the semantics of the language informally in the previous chapter. By discussing the metacircular interpreter we establish a more exact semantics. However, we will not discuss all aspects of the metacircular interpreter. Instead we will give an overview of the general structure and highlight some of the important aspects. Nevertheless the code listing of the full interpreter can be found in appendix A. A second goal is to use the metacircular interpreter as a means to explain how language constructs are added to the language. As mentioned above, a metacircular interpreter is an excellent means to experiment with new language constructs in a language. However, although the metacircular interpreter we discuss in this chapter runs properly we have not used it as a vehicle to experiment with language constructs because of performance issues. We wanted to be able to use AmbientTalk on small devices and still be able to

experiment and use language constructs on these devices. Therefore, we have placed reflective hooks in the native implementation of AmbientTalk to reconcile these requirements. Based on the metacircular interpreter we can identify these reflective hooks and explain how they behave.

This chapter is structured as follows. In the next section, we consider the general structure of the metacircular interpreter. in section 6.3 we discuss the interpretation of the passive object layer, followed by the discussion of the active object layer in section 6.4. Finally, section 6.5 deals with the introduction of the reflective hooks in the interpreter to enable experimentation with language constructs.

6.2 General Structure

The AmbientTalk interpreter is no different from most interpreters [AS96] and consists of a read-eval-print cycle. This cycle reads a text, converts it to a stream of tokens, which are matched to AmbientTalk's concrete grammar and converted to the abstract grammar specification by the parser. Evaluation of the abstract grammar is based on the reciprocity of evaluation and application, also a common pattern found in many interpreters based on a paradigm where processes are explicitly invoked in the program, such as functional-, procedural- or object-oriented style of computation. Note that in the remainder of this chapter we will have to refer to AmbientTalk from three different perspectives. First, we will talk about AmbientTalk as the *implementation language* for our metacircular evaluator. Second, we will refer to the *metacircular implementation* of AmbientTalk. Third, we will refer to AmbientTalk as the *language that is being interpreted* by the metacircular implementation. In the remainder of this chapter we will refer to the first as the *native* AmbientTalk, the second as the *metacircular* AmbientTalk and the third as the *base* AmbientTalk to avoid confusion. The relation between these different levels is shown in figure 6.1. The "circularity" stems from the fact that the semantics of the base AmbientTalk is identical to this of the native AmbientTalk.

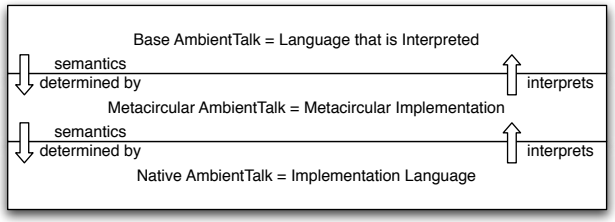

Figure 6.1: Connections between the different AmbientTalk perspectives

Since we implement the AmbientTalk evaluator in AmbientTalk and AmbientTalk features a prototype-based object model, it is a logical choice to represent the components of the metacircular evaluator in terms of objects and methods. The former are used to encapsulate the abstract grammar entities and the latter define how these entities are evaluated and applied to one another. An overview of all abstract grammar entities can be found in figure 6.2 on

page 156. The figure shows the hierarchy of objects that is created using nested mixin methods starting from the `AbstractGrammar()` mixin method. Mixin methods are drawn as a rectangular box, which is subdivided in two boxes. The upper part of the box contains the name of the mixin method with its formal parameters and the methods defined in the mixin method are placed in the lower part box. The `AbstractGrammar()` mixin method defines the protocol of the evaluation process through a number of abstract methods. The most part of this protocol is parameterized with an argument `e`. This is the context (or environment) of the evaluation process, represented as objects created from the `agContext` mixin, in which the expression is to be evaluated. This context of an evaluation consists of four objects:

1. `cur` refers to the current dictionary in which the evaluation takes place. As explained in section 5.3.3 a *dictionary* is AmbientTalk's first-class representation of a lexical environment that are used to represent objects. Dictionaries are represented as `agObject` objects. Receiver-less invocations are looked up in this dictionary.

2. `ths` refers to the dictionary of the original receiver of the invocation that is currently evaluated. (i.e. it points to the object returned by `this()`)

3. `sup` refers to the lexical parent dictionary of the invocation that is currently evaluated. A super send starts resolving these identifiers in this dictionary.

4. `thsActor` refers to the `metaActorBehavior` of the actor in which the current evaluation takes place and which is used to implement the reflective hooks. In the evaluation process it is used to retrieve the current actor its mail address, returned by `thisActor()`. An actor mail address is encapsulated in an `agActor` object. The `metaActorBehavior` itself is not shown in the structure, because it is not an abstract grammar entity. It is part of the behavior of the actor system itself and is further discussed in section 6.4.

In the structure of the abstract grammar entities we can distinguish between two types because an evaluator is a mapping from abstract syntax to "values":

- *syntactical entities*: these are internal representations of the abstract syntax tree nodes of AmbientTalk. They are represented by the mixin methods shown on the right hand side of the `AbstractGrammar()` mixin in figure 6.2. The most notable ones are the mixin methods `agReference`, `agTabulation` and `agApplication`. These mixins correspond to the columns of the syntax summary in table 5.10 on page 116, whereas the methods of these mixins correspond to the rows and implement the different evaluation rules.

- The runtime *value entities* are nested mixin methods of `agValue()`, which are self-evaluating abstract grammar components. The most interesting mixins are `agObject`, a representation for *dictionaries*; `agFunction`, `agClosure`, `agNative` and `agNativeClosure`, the representations of first-class (native) methods and finally, `agActor`, `agActorMessage`, `agActorBehavior`, `agMailbox` representations of the respective entities from the active object layer.

Now that we have given an overview of the structure and its important entities we can further discuss the double-layered object system in the next two sections.

6.3 The Passive Object Layer

In this section we discuss the semantics of the passive object layer. The structure of this section mirrors that of section 5.3, but we discuss the different topics from the perspective of its metacircular implementation.

6.3.1 Passive Objects

In section 5.3.3 we have seen that objects in AmbientTalk are represented as *first-class environments*, also called *dictionaries*. These dictionaries are represented as objects created from the mixin method `agObject`, which is shown in table 6.1. An object contains two linked lists of bindings (represented by `agBinding` objects), one for constants (`cst`) and the other for variables (`var`), and a pointer (`nxt`) to the next dictionary. Several methods are defined to search and manipulate these dictionaries. Because we have not focussed on efficiency, these lookup methods are rudimentary linear searches. That is to say, no optimization techniques, such as lexical addressing, have been implemented. As explained in section 5.3.3, an attribute is selected from an object using the dot-notation. In the metacircular evaluator this is evaluated by the `syncmessage` method in the different abstract grammar representations. The different implementations of this method, in `agReference`, `agApplication` and `agTable`, correspond to the different modes of qualifications summarized in table 5.10 on page 116, are shown in table 6.2. All implementations invoke `lookupConstant` on an `agObject`. This corresponds to the alignment of *definitions* and *declarations* with the visibility rules of an object that we have explained before. Indeed, as explained in section 5.3.3 a *declaration* corresponds to a *public* attribute whereas a *definition* corresponds to a protected attribute. Protected methods are invoked without the dot-notation in the context of an object and as a result such invocations are evaluated by the method `eval` in `agReference`, `agTabulation` or `agApplication`. This method, also shown in table 6.2, looks up the reference through a call to `lookupAny` on the dictionary, which in turn searches both constants and variables bindings corresponding to the visibility rules of a *protected* attribute.

6.3.2 Parameter Passing Semantics and Method Invocations

As discussed in section 5.3.2, AmbientTalk's methods feature three types of formal parameter lists: regular formal parameter table, variable argument lists (which we called *call-by-table*) and lazily evaluated variable argument lists (which we called *call-by-function-table*). These parameter binding semantics are realized in the metacircular implementation through an implementation of the method `call` in three abstract grammar entities: `agTable` and `agReference` and `agApplication`. These entities are, in the capacity as parameter bindings, representations in the abstract grammar of the concrete syntax for the different

```
agObject(cst,var,nxt) :: {
    ...
    containsConstant(nam, ths, e) :: { ... };

    lookupConstant(nam, ths, e) :: { ... };

    lookupAny(nam, ths, e)   :: {
         found: if ((v:cst.lookup(nam, ths,this(), e))~void,
                        var.lookup(nam,ths,this(), e),
                        v);
         if(found~void,
            if(nxt~void,
               Error("Could not find variable: ", nam.getTxt()),
               nxt.lookupAny(nam, ths, e)),
            found) };

    setVariable(nam, v) :: { ... };

    addVariable(nam, v) :: { var := agBindingP.cloneMe(nam, v, var); this() };
    addConstant(nam, v) :: { cst := agBindingP.cloneMe(nam, v, cst); this() };

    addFrame() :: this().cloneMe(agVoidP, agVoidP, this());
    ...
    capture() `<- agObject`
};
```

Table 6.1: Attribute Lookup in a dictionary.

```
agReference(name) :: {
    ...
    eval(e)      :: { v: e.cur.lookupAny(name, e.ths, e); v };
    syncmessage(dct, e) :: { dct.lookupConstant(name, dct, e) };
    supersend(e) :: e.sup.lookupAny(name, e.ths, e);
    ...
    capture() `<- agReference`
};

agApplication(expr, args) :: {
    ...
    eval(e)        :: expr.eval(e).apply(args, e);
    syncmessage(dct, e)::
      dct.lookupConstant(expr.getName(), dct, e).apply(args,e);
    supersend(e)   ::
      e.sup.lookupAny(expr.getName(), e.ths, e).apply(args, e);
    ...
    capture() `<- agApplication`
};

agTabulation(expr, idx) :: {
    ...
    eval(e)      :: expr.eval(e).get(idx.getTbl().eval(e).getMetaValue());
    syncmessage(dct, e) ::
      dct.lookupConstant(expr.getName(), dct, e).get(
        idx.getTbl().eval(e).getMetaValue());
    supersend(e) ::
      e.sup.lookupAny(expr.getName(), e.ths, e).get(
        idx.getTbl().eval(e).getMetaValue());
    ...
    capture() `<- agTabulation`
};
```

Table 6.2: Invoking a Message

```
agClosure(fun, env) :: {
   ...
   apply(args, e) :: {
     callframe : env.cur.addFrame();
     if(!args.isTable(),
        { args := args.eval(e);
          if(!args.isTable(),
             Error("Invalid actual arguments: "+args.print(e))) });
     fun.getPars().call(callframe,args,e);
     newE : agContext(callframe, env.ths, env.cur.parent(), e.thsActor);
     fun.getBody().eval(newE) };
    ...
};
```

Table 6.3: Application of a Closure

formal parameter lists that can be used to define the method. Hence, a particular `agTable` corresponds to a table of the formal parameters, i.e. `[a, b(), c]` in the method declaration `m(a, b(), c)::...`, which can in its turn contain *call-by-value* and *call-by-function* formal parameters; `agReference` corresponds to the reference `args` in `m@args:...`; and finally `agApplication` corresponds to the application `blocks()` in `m@blocks()::...`.

The method `call` is invoked in the `apply` method defined in `agClosure`, which is shown in table 6.3. The method extends the current lexical environment `env` of the closure and checks if the actual arguments list is a table. This is done to distinguish between the method *invocations* of the form `m(`a_1`, ..., `a_n`)` and `m@args`, which were also discussed in section 5.3.2. In the latter case the actual arguments are not a table, but an expression that evaluated to a table. Hence, this expression is first evaluated to a table. An error is thrown when the result is not a table. Subsequently, `call` is invoked on the formal parameter list of the function to bind the actual to the formal parameters and finally the body of the method is evaluated in the context of the new lexical context.

Now that we have discussed the semantics of a closure application, we can further detail the binding semantics of the formal parameter lists defined by the `call`. The different metacircular implementations of abstract grammar objects that deal with parameter binding semantics are shown in table 6.4 and are discussed below:

- The implementation of `call` in `agTable` loops over the table of formal parameters and calls the `bind` method on each formal parameter. The formal parameters in the table are either `agReference` or `agApplication` abstract grammar entities. The former its implementation of the `bind` method corresponds to a *call-by-value* binding and will bind the evaluation of the actual argument to the name of the formal parameter in the lexical environment of the function-call. The latter its `bind` implementation corresponds to a *call-by-function* formal parameter and therefore the actual argument is not evaluated but is used as the body of a newly created function with the formal parameter as its signature. This function is wrapped in a closure pointing to the lexical environment of the caller. Finally, this closure is used to extend the current lexical environment.

- `call` in `agReference` creates a new table `evaluatedActs` that contains all evaluated actual arguments. This table is bound to the name of the

```
agTable(tbl) :: {
    ...
    call(dct,actuals,e) :: {
       actT : actuals.getTable();
       forT : this().getTable();
       if (size(actT)!=size(forT), Error("non-matching argument list"));
       for(i:1,i<=size(forT),i:=i+1, forT[i].bind(dct,actT[i],e))
    };
   ...
};

agReference(name) :: {
    ...
    call(dct,actuals,e) :: {
      actT: actuals.getTable();
      i:0;
      evaluatedActs[size(actT)]: actT[i:=i+1].eval(e);
      dct.addVariable(name, agTableP.cloneMe(evaluatedActs)) };

    bind(dct,act,e) :: { dct.addVariable(name,act.eval(e)) };
    ...
};

agApplication(expr, args) :: {
  ...
  call(dct,actuals,e) :: {
    actT: actuals.getTable();
    i:0;
    closures[size(actT)]: agClosureP.cloneMe(
                              agFunctionP.cloneMe(expr,args,actT[i:=i+1]),
                              e);
    dct.addVariable(expr.getName(), agTableP.cloneMe(closures)) };

  bind(dct, act, e) :: {
    dct.addVariable(expr.getName(),
                    agClosureP.cloneMe(agFunctionP.cloneMe(expr,args,act),e)) };

  ...
};
```

Table 6.4: Evaluation of the different Types of Formal Parameters Lists in AmbientTalk

reference used to extend the current lexical environment.

- the implementation of `call` in `agApplication` creates a new table containing closures of functions that each have their body corresponding to the unevaluated actual arguments of the function-call. This table is used to extend the current lexical environment.

6.3.3 Mixin-Based Inheritance

In section 5.3.4 we have discussed the use of mixin methods as the model for introducing flexible delegation hierarchies that restrict the encapsulation problems resulting from inheritance. A mixin method is characterized by a call to the `capture` method as its last expression. As explained, this method returns the current dictionary, which represents the extension of the object onto which the mixin method was invoked. This is reflected in the implementation of the `capture` native:

```
captureNative(args, e) :: e.cur;
```

This dictionary is a proper extension of the current receiver as dictated by the method invocation semantics. That is to say, the next pointer (`nxt`) of the current dictionary, represented as an `agObject`) will point to the receiver of the mixin method. This is the result of invoking the mixin method itself. Such an invocation extends the receiver's dictionary with a new dictionary to represent the call frame for the invocation. This semantics is reflected in the metacircular implementation on the `agClosure.apply` method shown in table 6.3. This semantics also clearly shows that there is a unification between call frames and objects. That is to say a call frame is nothing more than an object that delegates to its receiver. Notice that such a semantics for method invocations is also present in the implementation of Self [US87].

Delegation is reflected in the metacircular implementation in the `supersend` methods, shown in table 6.2. These methods retrieve the parent object, to whom has to be delegated, from the context parameter `e.sup` and search for the attributes accordingly in both constant and variable bindings, corresponding to the semantics of *protected* attributes.

6.3.4 On Late-Binding and First-Class Methods

In section 5.3.5 we discussed the role of closures in the object model of AmbientTalk. Two of the important points was that when closures are created at lookup-time they support late-binding and can be used to represent first-class methods. In the metacircular implementation this semantics is reflected in table 6.5 in `agBinding.lookup`, which wraps the value resulting from the lookup into a closure with the current evaluation context. Remember from section 6.2 that the context consists of a pointer to the current dictionary, a pointer to the current receiver, a pointer to the lexical parent and the current actor. In this case the context of the new closure will have its current dictionary pointing to the object in which the attribute was found and it is also at this point that we set the lexical parent to the parent of the dictionary in which the attribute was found. Note that a method is wrapped with the context that consists of four environment pointers, as opposed to most functional languages where a function body is wrapped in a single lexical environment. The default implementation of the method `wrap`, implemented in the `AbstractGrammar` mixin method, is to return itself because wrapping only methods need to be wrapped in the context they are found. Hence, both the `agFunction` and `agNative` mixin methods override this method and return a closure respectively.

In section 5.3.5 we discussed that this scheme of creating closures at lookup-time has three advantages. We can now revisit these in the context of the metacircular implementation:

- A first advantage was that such a scheme enables late-binding polymorphism in the object model. This is reflected in `agBinding.lookup` where the `ths` variable is part of the context when an entity is wrapped. Hence, when a closure is applied it will have its *this* pointing to the correct receiver object in which this closure has been found. This pointer is changed when a closure with a different *this* pointer is applied. I.e. when an expression of the form `o.m(...)` is evaluated. This is reflected in table 6.2 in the implementations of the method `syncmessage` where the second argument of `dct.lookupConstant(..., dct, ...)`, which determines the

```
agBinding(nam,val,nxt) :: {
  ...
  lookup(n, ths, myDct, e) :: {
    if(n.getTxt()=nam.getTxt(),
       val.wrap(agValueP.agContext(myDct,
                                   ths,
                                   myDct.parent(),
                                   e.thsActor)),
       nxt.lookup(n, ths, myDct, e)) };
   ...
   capture() `<- agBinding`
};

agFunction(nam, pars, body) :: {
  ...
  wrap(e) :: agClosureP.cloneMe(this(), e);
  ...
  capture() `<- agFunction`
};

agNative(nat, nam) :: {
  ...
  wrap(e) :: agNativeClosureP.cloneMe(this(), e);
  ...
  capture() `<- agNative`
};

AbstractGrammar() :: {
  ...
  wrap(e) :: this();
  ...
  capture() `<- AbstractGrammar`
};
```

Table 6.5: Closures are Created at Lookup-Time

new `ths` pointer, is changed to the new receiver `dct`. This is in contrast to the lookup in the implementations of `supersend`, where the second argument of the lookup `e.sup.lookupAny(..., e.ths, ...)` refers to the current `ths` pointer.

- A second advantage is that reentrancy of methods in the context of object-based inheritance results from the creating closures at lookup-time rather than definition-time. At method definition-time `agFunction` objects are stored in the dictionary, which is shown in table 6.6. Indeed, the methods `define`, `declare` and `assign` store a clone of the `agFunctionP` prototype object in memory. Since `agFunction` objects are contextless they can be wrapped in different closures and be used in the context of a child or parent object. As a result the `agFunction` object needs to be allocated only once.

- The third advantage is that such closures enable first-class methods. Since `agClosure` is a first-class entity, it can be passed around. When the closure object is applied its body is evaluated in the context in which it was wrapped. In other words, in the context of the object from which it had been selected.

```
agApplication(expr, args) :: {
    ...
    define(exp, e) :: { e.cur.addVariable(expr.getName(),
                                          v:agFunctionP.cloneMe(expr, args, exp));
                        v.wrap(e) };
    declare(exp,e) :: { e.cur.addConstant(expr.getName(),
                                          v:agFunctionP.cloneMe(expr, args, exp));
                        v.wrap(e) };
    assign(exp, e) :: { e.cur.setVariable(expr.getName(),
                                          v:agFunctionP.cloneMe(expr, args, exp));
                        v.wrap(e) };
    ...
    capture() `<- agApplication`
};
```

Table 6.6: Functions, not Closures are Stored in a Dictionary

6.3.5 Cloning Objects

All the mixin methods that represent the abstract grammar are used to create prototypical objects. By convention, we have suffixed these prototypes with `P`. For example, `agClosureP` is the prototype object for a closure. These prototypical objects are then used throughout the metacircular implementation to create other objects by cloning them. This convention also ensures that behavior is reentrant such that is can be shared between all clones, because of the cloning semantics we explained in section 5.3.6. In that section we have discussed two native cloning methods `clone` and `copy` that rely on this semantics. This cloning semantics is reflected in the metacircular implementations of the method `picoClone`. It is shown in table 6.7. The implementation of `agObject.picoClone` returns a clone of the current dictionary with a reference to the constant bindings (`cst`), a shallow copy of the variables bindings (`var`) and the ancestors of that dictionary are recursively cloned until the dictionary `upTo` is encountered in the chain. The `agObject.picoClone` method reflects the cloning semantics of dictionaries, namely that declarations are reentrant and shared between cloned objects, since clones share the constant bindings, while a shallow copy of the variable bindings is taken. This semantics was discussed in detail in section 5.3.6 and illustrated by figure 5.3 on page 111. Hence, reentrancy doesn't occur because behavior is explicitly structured in modules or classes but rather results naturally from the semantics of cloning. This is in contrast to class-based languages, where behavior has been factored out in classes and as a consequence these classes are used as the basis to achieve reentrant behavior.

An advantage of reentrant behavior is that it can be shared between object such that less space is consumed to represent objects. In Self space-efficienct representations of objects are achieved through the use of maps [CUL89]. These maps rely on the fact that objects are often created by cloning them from another object as opposed to creating them ex-nihilo. Cloned objects have the same behavior and this behavior is factorized what is called a map. A map is an immutable structure that contains offsets for mutable data value slots and a map structure is shared between object that resulted from the clone. Self objects are internally represented as an array with a pointer to the map and the values are stored at each offset in that array. Maps are maintained internally

```
agVoid() :: {
    ...
    picoClone@args :: this();
    ...
    capture() ‘<- agVoid‘
};

agBinding(nam,val,nxt) :: {
    ...
    picoClone() :: cloneMe(nam,val,nxt.picoClone());
    ...
    capture() ‘<- agBinding‘
};

agObject(cst,var,nxt) :: {
    ...
    picoClone(upTo) ::
      this().cloneMe(cst,
                     var.picoClone(),
                     if(this()~upTo,
                        nxt,
                        nxt.picoClone(upTo)));
    ...
    capture() ‘<- agObject‘
};
```

Table 6.7: Clones

and are not visible at the language level. Hence, the semantics of maps in Self are similar to the reentrancy rules found in AmbientTalk its object model. Both object models have in common that reentrancy results from sharing constants and reentrancy naturally results from programming with prototypes. However, the internal representation of objects in Self is more efficient than AmbientTalk's object model because a map determines offsets for each slot such that an object can be represented as an array structure rather than a map structure. Hence, the slot names do not need to be stored in each object.

6.4 The Active Object Layer

Now that we have explained the most important aspects of AmbientTalk's passive object layer we can turn our attention to the semantics of the active object layer. Active objects are both the unit of concurrency and distribution as discussed in section 5.4. Hence, the straightforward choice to introduce parallelism in the metacircular implementation is to use native active objects. However, we could have designed the metacircular implementation in continuation passing style and use continuations to write our own scheduler and model active objects as different continuations. We opted for the former option, because this choice has several benefits:

- First, the option allows us to discover the semantics of true metacircularity in the context concurrency and distribution. A metacircular implementation in this context brings us to the question: "How can we express the ambient actor model in the model itself?".
- Secondly, a metacircular implementation allows us to demonstrate a sense of completeness with regard to the model of concurrency and distribution.

- Thirdly, since the model is expressed within itself it will allow us to abstract away from several technical issues and concentrate on the core semantics of the model.

Introducing the notion of actors in the metacircular implementation involves the representation of an *actor system*, as explained in section 4.2.2. The functionality of an actor system can be distilled from the ambient actor model we discussed in section 4.5, where an actor configuration was a formal representation for an actor system. The reduction rules on such an actor configuration specified the behavior of the actor system. From these reduction rules we can distill the main functionality of an actor system: actor creation and initialization, communication, actor address management, message evaluation and reification of both environment and communication. This functionality has to be present in the metacircular implementation and are explained in the following subsections.

Metacircularity gives a certain degree of freedom to choose what functionality we make explicit and for what functionality we will rely on the native AmbientTalk. For example, actor address management can be realized by maintaining lists of receptionists and external actors manually or we can rely on the metacircular implementation do this for us – in the passive object layer we also implicitly made such design choices. For example, the memory management in the metacircular implementation of the passive object layer relies on the garbage collector of the native implementation. Another example is that we did not make explicit the intrinsic details how methods invoke other methods and return values. An evaluator written in continuation-passing style provides more insight in these issues. For the metacircular implementation of the active object layer we made the design decision to take maximum advantage of the mechanisms for concurrency and distribution present in AmbientTalk, because it allows us to demonstrate the expressiveness of the model. The disadvantage of this choice is that such an implementation does not include all intrinsic details of the model up to the level of a realistic implementation. However, the most important details were already discussed as part of the operational semantics in section 4.5.

6.4.1 Actor Creation

Metacircular actors are represented as native actors that are initialized with a `metaActorBehavior`. Such a `metaActorBehavior` is in turn initialized with a metacircular passive object that determines the behavior of the base actor. The initialization of a metacircular actor is shown in table 6.8. The method `actorNative` determines the metacircular semantics for the method `actor`, used to create an actor in base AmbientTalk. This method evaluates its first argument, which is an expression that returns a metacircular passive object, and uses this object to initialize a new `agActor` object. This object represents a base actor *mail address* and is represented by a native actor mail address that was obtained from creating a native actor with `metaActorBehavior` as its behavior.

6.4.2 Structure of a Metacircular Actor

The `metaActorBehavior` object lies at the heart of the metacircular semantics of AmbientTalk's active object layer. Its structure is illustrated in figure 6.3 on

```
actorNative(args, e) :: {
  actorBehaviour: args.getTbl()[1].eval(e);
  agActorP.cloneMe(actorBehavior)
};

agActor(act) :: {
    getAct() :: act;
    setAct(anAct) :: act := anAct;
    cloneMe(anActorBehaviour) :: {
       act: actor(metaActorBehavior.new(anActorBehaviour));
       c: this().clone(root);
       c.setAct(act);
       act#initialize(c);
       c };
    isActor() :: true;
    getMetaValue() :: act;
    print(e) :: printBrackets(text(act));
    capture() `<- agActor`
};
```

Table 6.8: Metacircular Implementation of an Actor Address

page 157 and shows its most important attributes:

- a reference to a metacircular behavior `actorBehavior`
- a map of mailboxes `mailboxes`
- a map `mbxObservers` that contains the mailbox observers for each mailbox
- the message that is being evaluated `currentMessage`
- the default evaluation context for messages processed by this actor `context`
- the metacircular representation for a base actor mail address of this actor `self`

The mailboxes prefixed by `meta` are the mailboxes of the native actor. Next to these value attributes `metaActorBehavior` also contains a number of methods. The most important are `executeMessage`, which evaluates a metacircular message; `receiveMessage`, which corresponds to accepting the delivery of a metacircular message; `processNextMessage`, processes the next message in the mailbox.

An important part of the semantics of AmbientTalk's active object layer is determined by the semantics of the metacircular mailboxes, which realize and reify both the communication and the ambient of active objects. These mailboxes are represented in the code as `agMailbox` objects, shown in table 6.9. Metacircular mailboxes are not *directly* implemented in terms of the native mailboxes, because we want to make their semantics explicit. Instead, metacircular mailboxes are represented in terms of a vector. Nevertheless, metacircular mailboxes are linked to the native mailboxes through the use of the observer design pattern [GHJV94]. The metacircular mailbox is the subject to which the observers subscribe. However, in this realization of the design pattern the observers are *not* objects, implementing a `notify` method. Instead the observers are conceived as AmbientTalk's closures. There are two types of changes that can be observed. The first type of change is the addition of items to a mailbox and the

```
agMailbox(contents, addObservers, deleteObservers) :: {
  ...
  add(element) :: {
    contents.add(element);
    addObservers.iterate({ el(element) }) };
  delete(nr) :: {
    element: contents.delete(nr);
    deleteObservers.iterate({ el(element) }) };
  remove(element) :: {
    v: contents.remove(element);
    if(v, deleteObservers.iterate({ el(element) }));
    v
  };
  addSyncAddObserver(notify(el)) :: { addObservers.add(notify) };
  addSyncDeleteObserver(notify(el)) :: { deleteObservers.add(notify) };
  ...
  capture() `<- agMailbox`
};
```

Table 6.9: Implementation of Synchronous Observers based on Closures

second is the deletion of items from a mailbox. The former type of observers are subscribed to with `agMailbox.addSyncAddObserver` whereas the latter are subscribed to with `agMailbox.addSyncDeleteObserver`; each method has a call-by-function formal parameter `el`. Their implementation adds the closure bound to this parameter to the vectors `addObservers` or `deleteObservers`. These vectors of closures are iterated over and each closure is invoked whenever an element is added or deleted from a mailbox, respectively. Using this scheme mailboxes can be flexibly linked the metacircular mailboxes to the native mailboxes. These links are touched upon in the remainder of this section.

6.4.3 Mailbox Observers

Note the similarities and the differences between the mailbox observers we introduced at the native AmbientTalk in section 5.4.4 and the mailbox observers at the metacircular level. Both are a flexible realization of the observer design pattern, whereby one can subscribe first-class messages in the former case and closures in the latter case, such that the observers do not need to implement a fixed interface. The main difference between both is that mailbox observers at the native AmbientTalk are asynchronously invoked as opposed to the mailbox observers in the metacircular implementation which are synchronously invoked. The latter are synchronously invoked because they serve as a link that continuously synchronizes between the native mailboxes and the metacircular mailboxes. Synchronization between native and metacircular mailboxes ensures that, without having to implement metacircular mailboxes directly in terms of native mailboxes, the semantics of the metacircular mailboxes are translated to the semantics of the native mailboxes. For example, a message added to the `outbox` at the base AmbientTalk is to be translated to a message that is added to the `outbox` at the native AmbientTalk and this synchronous implementation of the observer design pattern realizes this. An asynchronous link between both types of mailboxes could result in incorrect semantics due to race conditions. This is illustrated by the following scenario. Consider that a message is removed from the `outbox` in a method executed by an actor running in the base

AmbientTalk. In the case of an asynchronous link between the native and the metacircular mailboxes, the message would be removed from the native mailbox after the base method has completed its execution. This is the case because an asynchronously invoked mailbox observer would remove the corresponding metacircular message from the native `outbox` after the method completed its execution. However, even though this first-class message is scheduled with priority (as explained in section 5.4.4) it can still be overrun by the communication process and as a result the message could be transmitted before it is deleted from the native mailbox. Therefore, it is important that metacircular mailboxes are fully synchronized with native mailboxes. Nevertheless, the mailbox observers that are reified in the base AmbientTalk are asynchronous as they were defined in section 5.4.4. To avoid confusion we will refer to the observers based on closures as *synchronous* mailbox observers, whereas the observers based on messages are called *asynchronous* mailbox observers.

The different synchronization links between the metacircular and the native mailboxes are explained throughout the remainder of this section. To avoid confusion the identifiers of the metacircular mailboxes are prefixed by `meta`. For example, if we write `metaInbox` then we refer to the metacircular implementation of the inbox. `inbox` refers to the inbox in the native AmbientTalk.

6.4.4 Processing Messages

The code related to message processing in the `metaActorBehavior` is shown in table 6.10. The method `initialize` subscribes a synchronous mailbox observer to the `metaInbox`. This observer asynchronously invokes `processNextMessage`. That method searches in the `metaInbox` for the first message that has a corresponding method implementation in `actorBehavior`. If such a message is found then the method `process`, defined in `actorBehavior`, is invoked with that message as its argument. Finally, if a message was found another `processNextMessage` is sent asynchronously. Such a recursive asynchronous message ensures that all messages that have a corresponding behavior are eventually executed by the active object. It is the incarnation of the perpetually running thread that every actor contains. The recursion stops when there are no more messages in the mailbox that have a corresponding method in the metacircular behavior and at that point the actor goes into a *sleep* state. There are two places in the code that will induce a transition to an *active* state again. First, when a new message is delivered then the method `receiveMessage` is activated and that method adds the metacircular message to the `metaInbox` and thereby triggers the synchronous mailbox observer that invokes the `processNextMessage` method. Second, the behavior can be replaced at the language level by means of the `become` method. This method has been implemented in the metacircular interpreter in terms of `setActorBehavior`. When this method is invoked then `processNextMessage` is also asynchronously invoked because the mailbox might contain messages that can be executed by this new behavior. Hence, this mechanism can be used to express conditional synchronization as explained in section 5.4.1.

```
metaActorBehavior::root.extend({
  ...
  setActorBehavior(anActorBehavior) :: {
    actorBehavior:=anActorBehavior;
    thisActor()#processNextMessage() };

  processNextMessage() :: {
    msgToProcessId: metaInbox.findFirst(
      actorBehavior.containsConstant(
        el.agActorMessage(context).getName(), actorBehavior, context));
    if(not(is_void(msgToProcessId)), {
       currentMessage:=metaInbox.get(msgToProcessId);
       metaInbox.delete(msgToProcessId);
       actorBehavior.agActorBehavior(context).process(currentMessage);
       thisActor()#processNextMessage()
    })
  };

  initialize(anAgActor) :: {
    self:=anAgActor;
    ...
    metaInbox.addSyncAddObserver({
      thisActor()#processNextMessage() });
    ...
  };

  receiveMessage(aMsg) :: {
    metaInbox.add(aMsg)
  };
  ...
});
```

Table 6.10: Code Corresponding to Processing Messages

```
metaActorBehavior::root.extend({
  ...
  initialize(anAgActor) :: {
    self:=anAgActor;
    ...
    metaOutbox.addSyncAddObserver({
      el.agActorMessage(context).getTarget().getAct()#receiveMessage(el)
    });
    metaOutbox.addSyncDeleteObserver({
      msg: el;
      idx: outbox.asVector().findFirst({
        and(el.getName()="receiveMessage", el.getArgs()[1]=msg) });
      if(not(is_void(idx)), outbox.delete(outbox.get(idx)))
    });
    sentbox.addAddObserver(thisActor()#onSentMsg);
    ...
  };

  onSentMsg(msg) :: {
    if(msg.getName()="receiveMessage", {
       metaMsg    : msg.getArgs()[1];
       metaOutbox.remove(metaMsg);
       metaSentbox.add(metaMsg) })
   };
  ...
});
```

Table 6.11: Code Corresponding to Message Delivery

6.4.5 Message Delivery

The metacircular code concerned with the asynchronous message delivery is shown in table 6.11. In the initialization of the `metaActorBehavior`, two synchronous observers are registered on the `metaOutbox`. One observer sends an asynchronous `receiveMessage` to the destination of the message that was added. Hence, a message sent in the base AmbientTalk results in a message in the `metaOutbox`, that in turn results in an asynchronous message sent at the native AmbientTalk. The other synchronous mailbox observer that is registered on the `metaOutbox` acts on a message that is deleted. In that case the native message `receiveMessage` is searched for in the `outbox` and is deleted if it is found. There are two ways in which a message can be deleted from an outbox. First, it can be deleted by manipulating the first-class mailbox in the base AmbientTalk. Second, a message is deleted from an outbox when an acknowledgment of the reception of the message has been received, as specified by the operational semantics of the ambient actor model we defined in section 4.5.4. This acknowledgment is intercepted in the native AmbientTalk by subscribing an asynchronous mailbox observer `onSentMsg` for additions on the `sentbox`. The method `onSentMsg` moves the messages for which it received an acknowledgment from the `metaOutbox` to the `metaSentbox`.

6.4.6 Asynchronous Message Passing

In section 5.4.2 we introduced explicit syntax for asynchronous message passing and message selection based on the `#`-operator. Both abstract grammar entities related to the evaluation of message passing and selection expressions are shown in table 6.12. The method `agReference.asyncmessage` represents the seman-

```
agReference(name) :: {
    ...
    `evaluate exp#name`
    asyncmessage(anActor, e) ::
      `return message(e.thsActor.getAddress(), anActor, name)`
      actorBehavior: e.cur.agActorBehavior(e);
      agActorBehavior.createMessage(
        e.thsActor.getAddress(), anActor, name, agVoidP) ;
    ...
    capture() `<- agReference`
};

agApplication(expr, args) :: {
    ...
    `anActor#name(args[1], ... args[n])`
    asyncmessage(anActor, e) ::
      `invoke send(aMsg)`
      actorBehavior: e.cur.agActorBehavior(e);
      aMsg: actorBehavior.createMessage(
        e.thsActor.getAddress(), anActor, expr.getName(), args);
      actorBehavior.send(aMsg) ;
    ...
    capture() `<- agApplication`
};
```

Table 6.12: Asynchronous Message Passing

tics for message selection and returns a metacircular message with the address of the current actor as its source. `agApplication.asyncmessage` represents the semantics for asynchronous message passing; it creates a metacircular message and invokes the method `send` with that message as its argument. This method will add the message to the `metaOutbox` of the current actor, which will in turn trigger the synchronous observer for additions on that mailbox that sends a native asynchronous message as explained above.

6.4.7 Reified Environmental Context

Reification of the environmental context in the metacircular implementation is achieved by a `discovery` actor, which is shared by all metacircular actors running in the same native AmbientTalk interpreter. In the `metaActorBehavior` of these actors there are a number of synchronous observers subscribed to the `metaProvidedbox` and `metaRequiredbox`, which are shown in table 6.13. These observers continuously update the `discovery` actor with information about what patterns are required and provided by actors running in the base AmbientTalk. If a matching pattern has been found by the `discovery` actor then it sends a `joinOn` message to the native actor representing the base actor that required the pattern. Similarly, if a pattern is no longer available the `discovery` actor sends a `disjoinOn` message to the native actor that represents the base actor that was previously joined.

The `discovery` actor provides and requires the same pattern `AmbientTalk` such that these actors can discover one another when they are in each others communication range. The `discovery` actor maintains a global structure that contains what patterns local actors provide and require. When another `discovery` actor is detected in its ambient then it transmits all the required patterns of its local base actors. The other `discovery` actors can then match

```
metaActorBehavior::root.extend({
  ...
  initialize(anAgActor) :: {
    ...
    metaRequiredbox.addSyncAddObserver({
      discovery#addRequiredPattern(self, el) });
    metaRequiredbox.addSyncDeleteObserver({
      discovery#removeRequiredPattern(self, el) });
    metaProvidedbox.addSyncAddObserver({
      discovery#addProvidedPattern(self, el) });
    metaProvidedbox.addSyncDeleteObserver({
      discovery#removeProvidedPattern(self, el) });
    ...
  };

  `callback message for discovery actor`
  joinOn(aPattern, providerActor) :: {
    idx: metaRequiredbox.findFirst(aPattern.getTxt() = el.getTxt());
    if(not(is_void(idx)), {
       metaJoinbox.add(agTableP.cloneMe([providerActor, aPattern])) })
   };

  `callback message for discovery actor`
   disjoinOn(aPattern, providerActor) :: {
    idx: metaJoinbox.findFirst({
      and(el.isTable(),
          el.getTbl()[1].getAct() = providerActor.getAct(),
          el.getTbl()[2].getTxt() = aPattern.getTxt()) });
    if(not(is_void(idx)), {
       resolution: metaJoinbox.delete(idx);
       metaDisjoinbox.add(resolution) })
  };
  ...
});
```

Table 6.13: Code Corresponding to Discovery in `metaActorBehavior`

these patterns and send the mail addresses of base actors that provide matching patterns. Since both `discovery` actors discover one another (because they both require and provide the "`AmbientTalk`" pattern) the same protocol is executed at both actors concurrently.

6.4.8 Concurrency Issues

In the previous chapters we have discussed three properties (discussed in sections 4.5.5 and 5.2.2) to avoid concurrency issues. We will now briefly discuss how these design principles are realized in the context of the metacircular AmbientTalk implementation:

- *Mailbox Privacy* states that mailboxes cannot be shared between actors. In the metacircular implementation this property is guaranteed, because metacircular actors are represented as native actors and mailboxes are implemented as passive objects. As a consequence of the containment principle, passive objects and thus mailboxes, are never shared between actors.

- *Serial mailbox access* states that a mailbox can only be manipulated by an actor or (exclusive) the actor system at the same time. The actor system manipulates mailboxes as a result of communication events such as the

transmission of a message. In the metacircular implementation presented above we have not made the transmission of messages entirely explicit. Indeed, there is no explicit communication component in the metacircular implementation. Instead actors communicate directly with one another via the native AmbientTalk interpreter. As a result the *serial mailbox access* property is inherited from the native AmbientTalk.

- the *Containment principle* states that no passive object can be shared by active objects. We have explained in section 5.2.2 that the containment principle was implemented at the native level through *call-by-deep-copy* parameter passing. Each time a parameter is passed over the boundaries of an active object it is deeply copied. In the previous chapter we have seen that there are two places where a parameter crosses an active object boundary. First, a new actor is created with a passive object to initialize the actor's behavior. This passive object is passed as *call-by-deep-copy* to prevent sharing between the newly created object and its creator. Second, when an asynchronous message is sent from one actor to another the message has to be deeply copied. Both actor creation and asynchronous message passing are translated to a native actor that is created and a native message that is sent. As a result, we can rely on the containment principle of the native AmbientTalk implementation.

Hence, in the metacircular implementation the properties all, either directly or indirectly, a result from the properties of the native AmbientTalk implementation. Note however, that this is a result from the design choice to take maximum advantage of the concurrency and distribution properties of the native AmbientTalk we made at the beginning of this section. For example, had we chosen to more explicitly represent the communication layer of AmbientTalk and encapsulated that layer into a hypothetical separate actor (henceforth called the *communication layer actor*) then the serial mailbox access property would not be inherited from the base-level AmbientTalk implementation. In that case all metacircular actor communication would be relayed through the communication layer actor. As a result, such a communication layer actor would have an internal representation for the outboxes of all actors. However, each base actor also has an internal representation of their outbox local to that actor. Since mailboxes are represented as passive objects they cannot be shared between a metacircular implementation of a base actor and the communication layer actor. As a result two copies of that mailbox would exist and would need to be consistently synchronized with one another each time they are manipulated. In the current metacircular implementation of AmbientTalk presented here we used synchronous mailbox observers (introduced in section 6.4.3) to synchronize native and metacircular mailboxes. However, with the introduction of a communication layer actor native and metacircular mailboxes would not always be local to the same actor anymore. Hence, similar race conditions to the ones explained in section 6.4.3 could occur. As a consequence, special care for the design of the mailbox synchronization protocol between the metacircular actors and the communication layer actor should be taken in order to preserve the serial mailbox access property, because both concurrently manipulate their representation of the same mailbox.

6.5 Reflection

In the previous chapter we have discussed the core of the AmbientTalk kernel and we have come to the conclusion that although the kernel implements the ambient actor model from the previous chapter it does not provide abstractions sufficient to structure application for mobile networks. In this section we will explore reflection as a means to define such abstractions. Reflection was explored in the context of object-orientation by Pattie Maes [Mae87] in the late eighties, but since then it has also been used as tool to structure concurrent and distributed applications [CM93, McA95, MMY96]. This section extends AmbientTalk with reflective operators. We have modeled reflection as a reification of implementation data structures at the metacircular level that are made available in the base AmbientTalk and absorbing base-level AmbientTalk objects back into the interpreter. This methodology of introducing reflection into a language was first proposed Smith [Smi82, Smi84].

AmbientTalk reifies part of its active object layer with the goal to serve as a language laboratory for facilitating experiments with ambient-oriented language constructs. We have chosen not to reify the passive object layer, because we want to focus on language constructs that deal with the coordination and interaction of actors. In the previous section we have discussed part of the semantics of AmbientTalk's active object layer in the form of the metacircular implementation. In this section we further detail this semantics in the same spirit, that is by means of an implementation in AmbientTalk. However, the semantics explained in this section is the parts that are reified and can be reflected on in the base AmbientTalk. The active object layer mainly consists of a protocol that defines how asynchronous messages are sent between two actors (that might reside on different machines). This protocol is also present in the metacircular implementation, explained in the previous section. Each stage in this protocol between the two interpreters is reified in the meta-object protocol (MOP) and is further discussed below. However, before we discuss this we detail how messages are reified and absorbed in the interpreter.

6.5.1 Reification and Absorption of Messages

In the metacircular implementation we have seen that a message is implemented as an `agActorMessage` object. Figure 6.2 shows that `agActorMessage` is in fact an object extension of an `agObject`. This reveals that a message is represented in the base AmbientTalk as an object. Indeed, the base-level representation of such a `message` prototype was shown in table 5.11 on page 117. Both `message` and `agActorMessage` have the same interface. The implementation of the `agActorMessage` object is shown in table 6.14. The implementation shows that `agActorMessage` is a wrapper that makes the base-level `message` object available in the metacircular AmbientTalk. For example, calling `agActorMessage.getSource` at the metacircular level will result in a `msg.getSource` call at the base-level if `msg` is the base-level `message` object delegated to by `agActorMessage`.

```
AbstractGrammar()::{
  ...
  agValue()::{
    ...
    agObject::{
      ...
      agActorMessage(context)::{
        sourceMethodName:: agTextP.cloneMe("getSource");
        targetMethodName:: agTextP.cloneMe("getTarget");
        nameMethodName  :: agTextP.cloneMe("getName");
        argsMethodName  :: agTextP.cloneMe("getArgs");
        setArgsMethodName  :: agTextP.cloneMe("setArgs");

        setContext(aContext) :: { context:=aContext };
        getContext() :: { context };

        getSource() :: {
          closure: this().lookupConstant(sourceMethodName, super(), context);
          closure.apply(agTableP, context)
        };

        getTarget() :: {
          closure: this().lookupConstant(targetMethodName, super(), context);
          closure.apply(agTableP, context)
        };

        getName() :: {
          closure: this().lookupConstant(nameMethodName, super(), context);
          closure.apply(agTableP, context)
        };

        getArgs() :: {
          closure: this().lookupConstant(argsMethodName, super(), context);
          closure.apply(agTableP, context)
        };

        setArgs(anArgsList) :: {
          closure: this().lookupConstant(setArgsMethodName, super(), context);
          closure.apply(agTableP.cloneMe([ anArgsList ]), context)
        };

        capture() `<- agActorMessage`
      };
      ...
    }
    ...
  }
  ...
}
```

Table 6.14: Absorption of a Message in the Metacircular AmbientTalk

6.5.2 Reification and Absorption of Actor Communication

An actor deals with messages in three ways: first, it must be able to create messages; secondly it must be able to send messages and finally an actor must process messages it received. These three actions have been reified in the base AmbientTalk using a similar scheme as for the reification of messages. In figure 6.2 we see that `agActorBehavior` is an object extension of an `agObject`. The former represents a wrapper for metacircular passive objects that are used as a behavior for an active object, similar to how `agActorMessage` objects are wrappers for base-level `message` objects. Such base-level passive objects have an implementation for the methods `createMessage`, `send` and `process`. Their use and default implementations are discussed below.

Message Creation

In section 6.4.6 we have discussed two instances in the metacircular implementation where messages are created by an actor. This was reflected in the metacircular code in table 6.12. Both the evaluation of a message selection expression (`act#m`) and a message passing expression (`act#m(`a_1`, ..., `a_n`)`) results in a `message` object that is created by calling `createMessage` on an `agActorBehavior` object. The default implementation for `createMessage`, included in the `root` object, is shown below.

```
createMessage(src, target, name, args)::message.new(src, target, name, args)
```

This method can be regarded as a factory for messages, sent by an active object. The method is looked up in the passive object that defines the behavior of that actor. Hence, it is possible to override this method in an actor behavior such that another implementation of a message is returned and used as part of the communication process. This is a common technique which will be used frequently for supporting the introduction of ambient-oriented language constructs.

Message Sending

The second part of the communication process that has been reified in the base AmbientTalk is message sending. This was also discussed in section 6.4.6 as part of the metacircular implementation of message passing, defined in the method `asyncmessage` of the `agApplication` mixin (code shown in table 6.12). A message is sent by means of a call to the method `send` defined in `agActorBehavior`. The default behaviour of message sending between actors is defined by the following method defined in the `root` object:

```
send(msg)::{
  outbox.add(msg);
  void
}
```

An expression of the form `anActor#msg(`a_1`, ..., `a_n`)` is evaluated in terms of a call to the meta-level method `send`. For example, the expression `mycounter#increment()` is translated by the AmbientTalk interpreter to the call `send(createMessage(thisActor(), mycounter, "increment", []))`.

As illustrated below, it is possible to override the default behaviour by redefining the method `send` in the actor behavior. The example above displays

"before send" and "after send", respectively before and after sending the message.

```
send(msg)::{
  display("before send", eoln);
  .send(msg);
  display("after send", eoln)
}
```

Message Processing

Messages are processed at the metacircular level in `metaActorBehavior` by the method `processNextMessage`, shown in table 6.10. `processNextMessage` checks if the current actor behavior has a method definition for messages in its `inbox` and if it is the case the `process` method is invoked on the `agActorBehavior` wrapper. The default behaviour of an actor for processing a message is defined by the following method in the `root` object in the base AmbientTalk:

```
process(message)::{
  execute(message)
}
```

The method `execute` is part of the `root` object. It invokes the method associated with the name of the message in the behaviour of the actor.

The `process` method can be overridden just like the `send` method. Note that the reflective operators `send` and `process` are aligned with the communication mechanisms of the AAM. The `send` and `process` methods are implemented in terms of the mailboxes `outbox` and `inbox` respectively. This is important in the context of the preservation of the non-blocking characteristic of the AmOP paradigm, which is discussed in the next section.

6.5.3 Mailboxes in the Context of Reflection

In section 5.4.4 we have discussed the design of first-class mailboxes in the kernel and the use of *mailbox observers* to monitor changes. We will now consider the use of first-class mailboxes again in the context of reflection. A metaprogram based on mailboxes consist of two types of actions, namely *reification* and *reflection* of communication events.

Reification

Reification of communication events is achieved through the use of *mailbox observers*. More particularly, we can use of mailbox observers to observe both additions and deletions to mailboxes. Depending on the mailboxes that are observed we can place hooks to intercept different communication events of the actor system. To illustrate this, suppose that the methods `onIncomingMsg` and `onProcessedMsg` are subscribed as mailbox observers for additions to the `inbox` and `rcvbox` respectively.

```
onIncomingMsg(msg)::{ display("received ", msgName(msg), eoln) };
onProcessedMsg(msg)::{ display("processed ", msgName(msg), eoln) }
```

Since mailbox observers are scheduled with priority (discussed in section 5.4.4), the `onIncomingMsg` method will be executed before the message it observed is executed. As a result, these mailbox observers enable metaprograms that

are based on events of the communication between actors. The notification of changes in the mailboxes occurs in an asynchronous fashion so that this mechanism can be used to write event-based asynchronous metaprograms in the language as illustrated by the two toy mailbox observers above.

The use of mailbox observers on an `inbox` is an alternative way to reify low-level message reception in AmbientTalk's meta-level interface. Message reception is often introduced in meta-level architectures designed for distribution and concurrency, such as CodA [McA95] where it is reified by the `accept` method. In CodA the `accept` method is used to intervene in the synchronization and transmission of messages at the meta-level and concerns the interaction between the sender and the receiver. The introduction of an `accept` method allows a developer to design blocking communication abstractions. For example, the `accept` method can be redefined such that a message is executed instead of being placed in the `inbox`. This is in contrast to the mailbox observers found in AmbientTalk. Mailbox observers give the flexibility to intervene in the synchronization and communication process at the meta-level, while precluding the implementation of blocking communication primitives. The reason for this is that AmbientTalk's meta-level interface reifies actor communication through an asynchronous event based system such that messages are always placed in the `inbox` before they are processed. Note that registering a mailbox observer for additions on the `inbox` that immediately processes the message is the equivalent of overriding an `accept` method that processes the message immediately. However, since the message is placed in the inbox first the message will be executed independently of the control flow of the sender.

Reflection

Reflection of the communication events occurs through the direct manipulation of mailboxes. A mailbox is most frequently manipulated through the methods `add`, `delete` and `get`, which allow one to add, remove and retrieve elements respectively. These methods enables metaprograms to intervene in the default communication process amongst active objects. For example, `outbox.delete(msg)` removes a message `msg` from the `outbox`, thereby preventing it from being transmitted to another actor. Note that the reflection of the communication events will never result in race conditions with the actor system, because of the properties we have discussed in sections 4.5.5 and 6.4.8 in the context of both the ambient actor model and the metacircular implementation. We will now illustrate the use of both reification and reflection mechanisms to introduce an alternative behavioral synchronization mechanism.

Example

Consider the implementation of a synchronized bounded buffer in table 6.15, which is based on enabled-sets that were introduced by Tomlinson and Singh [TS89]. An enabled-set defines the messages, depending on the state of the object, that may be processed. For example, an empty buffer (denoted by the state `empty`) can process `put` messages, but no `get` messages. Conversely, when there is no more space available for elements, the buffer is `full` and can only process `get` messages and finally, when the buffer is neither full nor empty it is in the `intermediate` state and can process all messages. The active enabled-set

```
bufferBehaviour: root.extend({
  MAX      :: 3;
  ptr       : 1;
  queue[MAX]: void;
  enabledSet: vector.new();
  delayedMbx: mailbox.new("delayed");
  empty        : [put, init];
  intermediate : [put, get, init];
  full         : [get, init];

  enable(methods)::{
    enabledSet:=
      vector.newWithTable(methods);
    for(i:1, and(i<=delayedMbx.length(),
                 not(enabled)), i:=i+1, {
      msg: delayedMbx.get(i);
      if(isEnabled(msg), {
           enabled:=true;
           delayedMbx.delete(msg);
           inbox.add(msg) })
    })
  };

  isEnabled(msg)::
    enabledSet.detect({
      msg.getName() = getMethodName(el) });
```

```
  put(el)::{
    queue [ptr] := el;
    ptr:=ptr +1;
    if(ptr >MAX,
       enable(full),
       enable(intermediate))
  };

  get(el)::{
    ptr:=ptr - 1;
    client#result(queue[ptr]);
    if(ptr = 1,
       enable(empty),
       enable(intermediate))
  };

  onIncomingMsg(msg)::{
    if(not(isEnabled(msg)), {
       inbox.delete(msg);
       delayedMbx.add(msg) })
  };

  init()::{
    inbox.addAddObserver(
      thisActor()#onIncomingMsg);
    enable(empty) }

})
```

Table 6.15: Bounded Buffer

can be changed with the method `enabled`.

The synchronization is handled by the `inbox` observer for additions named `onIncomingMsg`. This observer checks if newly received messages are in the active enabled-set. If this is not the case then the message is moved from the `inbox` to a custom mailbox named `delayed`. In the other case no action is undertaken such that the message remains in the `inbox` and is processed accordingly. The active enabled-set is changed with the `enable` method. After such a change it is possible that messages that were previously delayed can now be processed. For this reason the `enable` method iterates over the `delayedMbx` mailbox and moves enabled messages back to the `inbox` such that they are processed.

6.5.4 Discussion

In the previous subsection, we have considered the use of mailboxes for reifying and reflecting upon the communication processes in the AmbientTalk kernel. The same mechanism of mailboxes and mailbox observers can be used for the reification and reflection on the ambient of the AmbientTalk kernel. In the previous chapter, we have described how the mailboxes `requiredbox`, `providedbox`, `joinbox` and `disjoinbox` are used to reify the environmental context of actors. However, the reflective capabilities on the reified ambient are less expressive than the reflective capabilities of the communication process. The reflection of the communication process makes it possible to intervene and change the communication process, thus enabling *intercession*. In contrast, the reification of the ambient cannot change the environment directly, we can merely observe

the environment and act upon these observations. To illustrate the difference, it is possible to remove a message from the `outbox` thereby avoiding that it gets transmitted, while removing a resolution from the `joinbox` will not remove the discovered actor from the environment.

When designing a meta-level architecture one needs to be careful to avoid infinite regression. Consider an active object with an inbox observer for additions. When a message is received by that active object, then the inbox observer message is placed in the `inbox`, this will in turn cause an observer message to be placed in the `inbox` and so forth. In all, the observer messages are observing the observer messages themselves and so forth, which is causing an infinite regression. To prevent such an infinite regression mailbox observers do not observe themselves.

6.6 Composition of Metaprograms

Based on the MOP explained in the previous section it is possible to write metaprograms. However, we have not yet discussed how metaprograms are composed together. To facilitate composition of metaprograms it is important that metaprograms can be *encapsulated* and can be *separated* from the base-level [BU04]. When we consider the code shown in table 6.15 again, it is clear that it does not fulfill these encapsulation and separation properties because the code of AmbientTalk's MOP is not contained in separate module and is tangled with the base-level code. In AmbientTalk we can use mixin methods for both encapsulation and separation of metaprograms. Table 6.16 shows the meta-level code of the enabled-set from table 6.15 factored out and encapsulated in a mixin method. Mixin methods that encapsulate an extension of the MOP are henceforth called *meta-mixins*. An important advantage of encapsulation is information hiding, which enables the interchangeability of modules. Meta-mixin methods support such interchangeability, for example it is possible to reuse the bounded buffer, shown in table 6.15 with a different implementation of an enabled-set mixin on the condition that it supports the same interface, which is the common rule to enable interchangeability in the object-oriented paradigm.

Mulet et al. [MMC95] already noticed that the use of mixins could provide an expressive means for composition of meta-objects. Classic inheritance has the disadvantage that, when used as a means to compose meta-classes together, they must fix the order of the inheritance hierarchy. Another disadvantage of classic inheritance is code duplication if the same code is to be reused in the context of multiple parent classes. Meta-mixins do not have these disadvantages, they benefit from the same type of flexibility that is achieved with base-level mixins, which was discussed in section 5.3.4. Hence, meta-mixins can flexibly choose the linearization order of different mixins at run-time. This flexibility enables that the different meta-mixins can be reused in different contexts. Nevertheless, it is necessary to carefully design the meta-mixins with reuse in mind.

6.6.1 Implementing Language Constructs using Meta-Mixins

Above, we have discussed mixins as the unit of encapsulation for changes to the meta-layer. However, if we consider the meta-mixin for enabled-sets, shown

```
enabledSetMixin()::{
  enabledMethods: vector.new();
  delayedMbx    : mailbox.new("delayed");

  ‘public part of language mixin‘
  enable(methods)::{
    enabledMethods:=
      vector.newWithTable(methods);
    enabled: false;
    for(i:1, and(i<=delayedMbx.length(),
                 not(enabled)), i:=i+1, {
      msg: delayedMbx.get(i);
      if(isEnabled(msg), {
         enabled:=true;
         delayedMbx.delete(msg);
         inbox.add(msg) })
    })
  };
```

```
  ‘private part of language mixin‘
  isEnabled(msg)::
    enabledMethods.detect({
     msg.getName() = getMethodName(el)
    });

  onIncomingMsg(msg)::{
    if(not(isEnabled(msg)), {
       inbox.delete(msg);
       delayedMbx.add(msg) })
  };

  capture()
};
```

Table 6.16: Language Mixin for EnabledSets

```
bufferBehaviour: root.extend({
  MAX      :: 3;
  ptr       : 1;
  queue[MAX]: void;

  put(el) :: {
    display("put", eoln);
    queue [ptr] := el;
    ptr:=ptr +1
  };

  get(client) :: {
    display("get", eoln);
    ptr:=ptr - 1;
    client#result(queue[ptr])
  };

  syncMixin()::{
    put(el)::{
      .put(el);
      if(ptr >MAX,
         enable(full),
         enable(intermediate))
    };
```

```
    get(el)::{
      .get(el);
      if(ptr = 1,
         enable(empty),
         enable(intermediate))
    };

    empty        : [put, init];
    intermediate : [put, get, init];
    full         : [get, init];

    init()::{
      inbox.addAddObserver(
        thisActor()#onIncomingMsg);
      enable(empty) };

    requires(enabledSetMixin);
    capture()
  }
});

bb: actor(bufferBehavior
          .enabledSetMixin().syncMixin())
```

Table 6.17: Bounded Buffer Redesigned with Mixins

in table 6.16, then we can see that it does not only override the MOP, but also introduces a method `enable` that interacts with the behavior defined in the meta-mixin. Hence, this meta-mixin does not only redefine the MOP but also provides an interface that is to be used from the base-level to steer the behavior of the meta-level. In AmbientTalk we consider such meta-mixins as language constructs, because as opposed to most meta-programs they do not remain transparent to the base-level. Meta-mixins that are used to introduce language constructs are called *language-mixins*. Reflection has been explored before [MMY96] in the context of adding parallel and distributed language constructs to an existing language. This mechanism of introducing language constructs into the AmbientTalk kernel based on reflective modifications is important because it facilitates easy experimentation with language constructs for ambient-oriented programming.

6.6.2 Scoped Reflection

When extending the semantics of the kernel through its MOP, it is sometimes useful to scope the effects of these changes. For example, if we want to employ reflection for debugging purposes we want to collect only relevant information with respect to the bug. There are three reasons why scoped reflection is particularly useful in the context of our research:

- Scoped reflection is useful is the introduction of language constructs through the MOP. Many language constructs delimit their application scope. For example, the `synchronized` keyword in Java delimits a method or block of statements.
- Scoping the reflection is also interesting from the perspective of integrating different language concepts. When designing a new language one has to be careful not to fall into the trap of *feature piling*, rather before adding a new concept to a language one has to think if it is possible to *unify* existing concepts of a language to realize the same semantics of that feature [Hoa73]. To support this unification approach of language design one needs to be able to experiment with the integration of language concepts. In AmbientTalk new language concepts can be introduced reflectively. Hence, in AmbientTalk the integration of concepts translates to the integration of reflective changes. A reduced scope of these changes thus facilitates the integration of language concepts and their associated language constructs because their interactions can be more easily anticipated.
- Another advantage for scoping the reflection is that locality permits economy in the use of computational resources as Hoare also noted in his paper. Indeed, if a language feature is not used its impact on resources should be limited or nil.

In this subsection we discuss four different techniques to scope the effects of metaprograms.

Global and Active Object Scope

The default reflective operators presented in this chapter reside in the `root` object. The reflective operators can be redefined in the same root object (using

assignment) or locally overridden in a particular object extension through the use of mixin methods. In the former case, which as called global scope, the redefined reflective operators will change the behavior of all newly defined actors in the system. In the latter case, which is called active object scope, the redefined reflective operators will only affect the actors defined from that object extension. Hence, in the former case the changes will be global, while in the latter case the reflective extensions will be restricted to the active object itself. However, actors will not be affected by changes to the `root` object made *after* they have been initialized, because actors are initialized with a deep copy of the passive object to enforce the containment principle explained in section 5.2.2. Global scope is mainly used to configure AmbientTalk with a particular extension, which avoids the burden of having to apply the same MOP extensions to all objects. Active object scope on the other hand allows one to flexibly experiment and configure active objects with a redefined MOP according to their requirements.

Message Scope

Based on the MOP presented above we can create a design pattern to make the extensions of the AmbientTalk kernel local to the scope of a single message sent from one active object to another. The rationale of the design pattern is to involve messages in the processing semantics. Table 6.18 (left) shows the code of a meta-mixin based on this principle. This meta-mixin overrides the `createMessage` such that it returns an extension of a `message` object that contains a method `process` which takes the behavior of the active object as its argument and in turn calls `execute` to execute the method associated to itself. The mixin also overrides the `process` method in the active object behavior to pass on the responsibility of processing the message to the message itself. Once that we have applied this message-scope mixin to the behaviors of active objects, it is possible to create extensions of the *message scope* such that it contains behavior which will be executed along the control flow of a message. For example, table 6.18 (right) shows an implementation of such a meta-mixin. This mixin redefines the `process` method of the message such that acknowledgments are sent back to the sender once a message has been processed by its receiver. Note how both the `messageScopeMixin` and `ackMixin` make use of dynamic object extensions. In the former case the message returned by the super-call to `createMessage` is *dynamically* extended to include the `process` method and in the latter case a further extension of such a message is made such that this `process` method is redefined to send an acknowledgment message. Hence, with each call to `createMessage` in the delegation chain the existing message is further augmented with behavior so that it finally has all the necessary behavior.

Also note that only the sender (and not the receiving actor) needs to be extended with the `ackMixin`. However, both the sending and receiving actor need to have the `messageScopeMixin` applied, unless one would initialize the AmbientTalk kernel with this mixin applied to the `root` object. Nevertheless, once the `messageScopeMixin` is applied to the receiver each message that is received can individually determine how it should be processed.

```
messageScopeMixin()::{

 createMessageScope(aMessage)::{
   aMessage.extend({
     process(behavior)::{
       behavior.execute(this())
     }
 });

 createMessage(src, target,
               name, args)::{
   createMessageScope(
     .createMessage(src,target,
                    name,args))
 };

 process(msg)::{
   msg.process(this())
 };

 capture()
}
```

```
ackMixin()::{
  createMessage(src, target, name, args)::{
    .createMessage(src, target, name, args)
    .extend({
      process(behavior)::{
        value: .process(behavior);
        getSource()#ack(this());
        value
      }
    })
  };

  ack(msg)::{
    ui#display("message ",
               msg.getName(),
               " has been received.", eoln)
  };

  requires(messageScopeMixin);
  capture()
}
```

Table 6.18: Message Scope Mixin (left) and an Acknowledgment Mixin based on Message Scope (right)

Language Construct-Based Scope

Another type of scoping the reflection is based on the notion of a language construct. In section 6.6.1 we have discussed that meta-mixins can be used to define language constructs. These language constructs are represented as methods that configure the redefined parts of the MOP. When such methods are combined with the call-by-function parameter passing semantics, explained in section 5.3.2, we can introduce language constructs that embed an expression or a group of expressions. An example of such a language construct in Java is the form `synchronized(aLockObj)` { `...` `statements` `...` }, where the statements in the block are atomically executed in the context of a specific lock. To introduce this type of language constructs in AmbientTalk from within AmbientTalk itself it is useful if part of the MOP behavior can be overridden *in the context of the language construct* so that the expressions are evaluated with respect to the redefined behavior of that MOP.

We illustrate this type of scoping with a simple meta-mixin shown in table 6.19. The language-mixin introduces a language construct `log`, which takes a call-by-function argument `block`. This construct overrides both `createMessage` and `send`. Note that both methods are overridden in the context of the `log` method and not in the meta-mixin. Finally, the formal parameter `block` is invoked in the dynamic scope of the language construct.

This type of scoping can be regarded as a form of local dynamic scoping of the MOP. The dynamic scope combined with the invocation semantics of AmbientTalk's object model will cause `block` to be evaluated in the correct context. Since `block` is called with `dynamicScope` its scope will be the `log` construct, instead of the scope of the caller. The `log` construct its scope will be the context of its caller, because it was called without a qualification. Remember that invocations without a qualification are invoked in the context of the current scope

```
languageConstructScopeMixin(logger)::{
  notifyLogger@strings::{
    logger#display@strings
  };

  log(block())::{

    createMessage(s, t, n, a)::{
      super().notifyLogger("before createMessage ", n, eoln);
      msg: .createMessage(s, t, n, a);
      super().notifyLogger("after createMessage ", n, eoln);
      msg
    };

    send(msg)::{
      super().notifyLogger("before message sent", eoln);
      .send(msg);
      super().notifyLogger("after message sent", eoln)
    };

    `execute block in the current evaluation context`
    dynamicScope(block)()
  };

  capture()
}
```

Table 6.19: Example: Mixin using Language Construct Based Scope

MOP overridden/refined in	Scope
`root` object	global
object behavior	all actors using that object as behavior
method	language construct-based scope (using local dynamic scoping)
message	messaging protocol handling this single message

Table 6.20: Summary of the different Reflective Scopes

as a result of the dynamic closure creation scheme, explained in section 5.3.5. Hence, `block` will be called in the scope of the overridden MOP chained to the the scope of its caller.

The use of the `log` construct is shown below:

```
counter: actor(counterBehavior);

log({
  counter#increase();
  counter#decrease()
})
```

It illustrates that the scope of the expressions used in the language construct can still refer to the scope outside of the language construct.

6.7 Conclusion

In this chapter we discussed the semantics of both the passive and active object layers on the basis of a metacircular implementation. Based on this semantics

we have introduced a MOP into AmbientTalk. The MOP was designed to allow one to redefine the communication processes of actors. We have discussed two types of changes to the MOP. First, the methods (`createMessage`, `send` and `process`) that interface with the AmbientTalk kernel language can be overridden and redefined. Second, through the use of mailbox observers one can observe changes made to the mailboxes and act on these changes. Next to the definition of this MOP we have also discussed how these metaprograms can be introduced gracefully with respect to some design principles. These design principles can be applied through the use of meta-mixins. These meta-mixins can be flexibly composed and allow one to introduce language constructs into the kernel. Next to the composition possibilities we have also discussed several approaches to scope the effects of the changes to the MOP. These approaches are summarized in table 6.20.

The reflective capabilities that we introduced in the AmbientTalk kernel play a vital role in the remainder of this dissertation. They facilitate experiments language constructs for ambient-oriented programming and allow them to be reused in different AmOP applications. In the following chapters we discuss these experiments with language constructs.

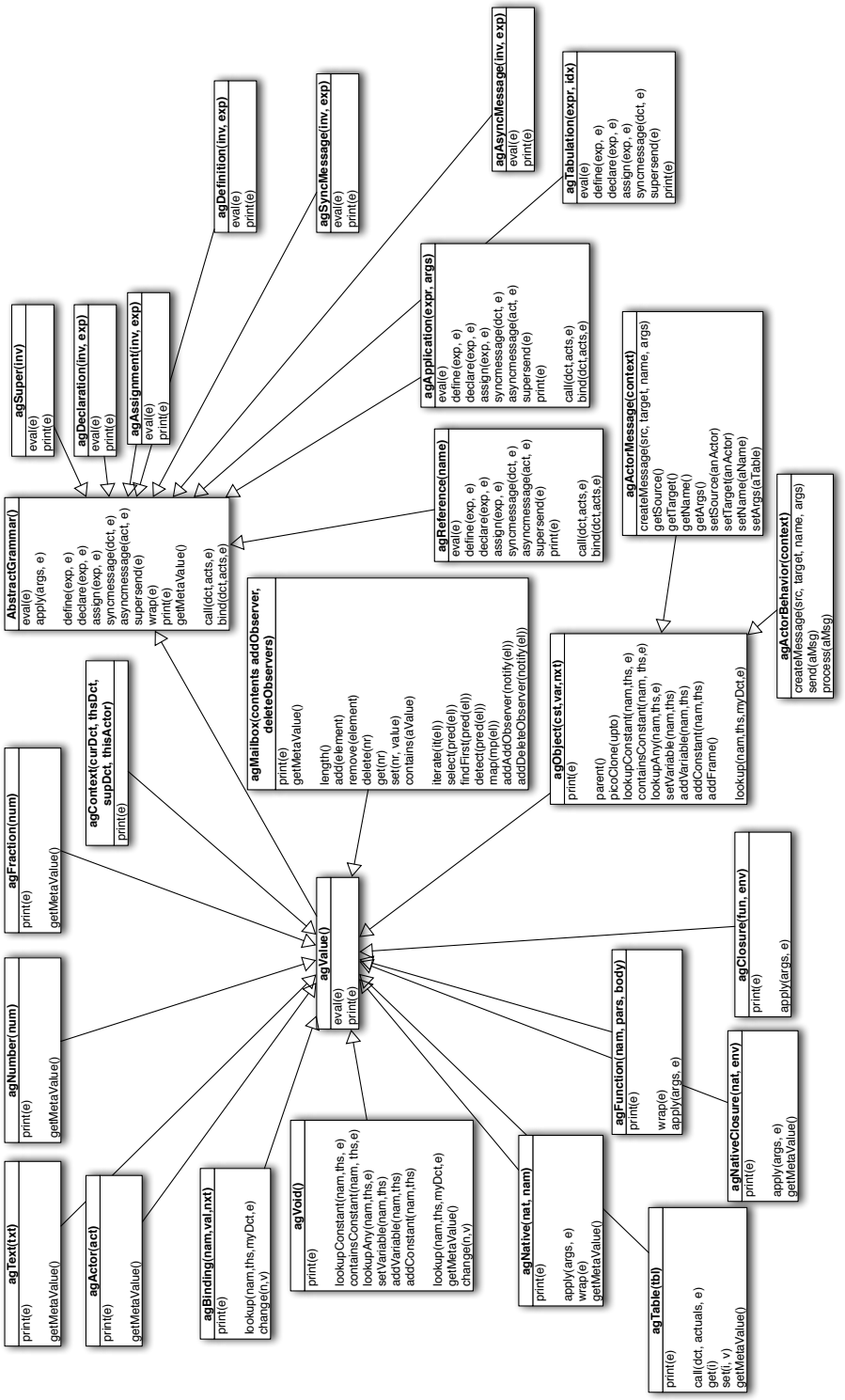

Figure 6.2: Abstract Grammar of the Metacircular Interpreter

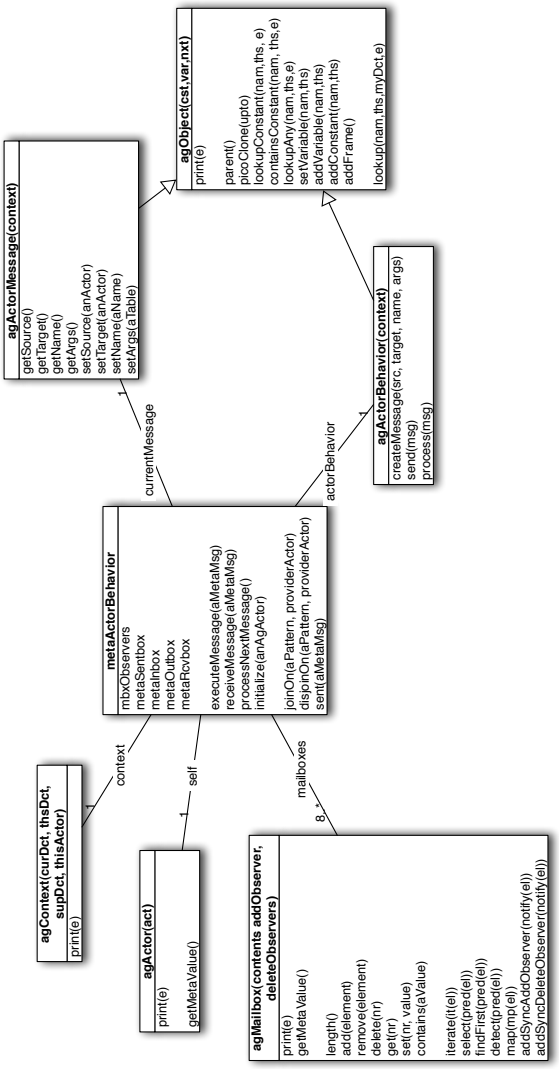

Figure 6.3: Structure of `metaActorBehavior`

Chapter 7

AmbientTalk at Work: Ambient-Oriented Language Constructs

In the previous chapter we have addressed two aspects of AmbientTalk. First, we have defined the semantics of AmbientTalk through a metacircular implementation. Second, we have used this metacircular interpreter to define reflective hooks that make it possible to define language extensions. The latter step enables the use of AmbientTalk as a language laboratory for mobile distributed systems. Given this setup, we can now address our first and second research goal (discussed in section 1.1.3), namely to (1) uncover suitable language constructs that deal with the hardware characteristics we defined in section 2.3 and (2) gain insight in the structure of AmOP applications.

7.1 Introduction

This chapter introduces a number of language constructs to explicitly deal with the hardware phenomena encountered in mobile distributed systems. We have adhered to the (functional programming) tradition of modular interpreters to formulate these features as modular semantic building blocks – called *language mixins* – that enhance AmbientTalk's kernel. This methodology facilitates our experimental approach and allows us to easily define a new AmbientTalk flavor composed out of different combinations of language constructs.

The first part of this chapter addresses our research goal to discover language constructs that deal with the hardware phenomena encountered while constructing mobile distributed systems. In the next section we introduce a number of synchronization and coordination abstractions to deal with the consequences of the natural concurrency of the autonomous devices. These constructs are based on existing synchronization constructs found in the state of the art. In section 7.3 we address the ambient resources hardware phenomenon and introduce special types of distributed object references, called ambient references. These references encapsulate both the discovery of and communication with ambient resources. As a result we can more easily structure AmOP applications that rely

on ambient resources. Finally, in section 7.4 we present a language construct to deal with the consequences of long-term disconnections. More particularly a language construct is defined to specify customized message delivery strategies based on the notion of timeouts. These timeouts can be used to deal with application-specific issues related to message deadlines.

The second part of this chapter addresses our research goal to gain insight in the structure of AmOP application. In section 7.5 we discuss two similar AmOP applications. One application is developed in Java whereas the other is developed in AmbientTalk using some of the language constructs explored in this chapter. Subsequently, we compare both applications with respect to their capabilities to deal with the hardware phenomena discussed in section 2.3.

7.2 Synchronization and Coordination

As discussed in section 2.3, one of the consequences of the autonomy of devices is that their interactions are naturally concurrent. Concurrency implies a need for synchronization such that the concurrent actions are meaningful and do not lead to inconsistencies. This section discusses a number of language constructs the aim of which is to coordinate the actions of various active objects that are concurrently performing tasks. Most of the language constructs presented in this section are based on existing abstractions which have proven their merits in traditional distributed systems.

7.2.1 Guards

Introduction

In section 6.6 we discussed the language-mixin that introduced the *enabled-sets* language construct (table 6.16 on page 150) into AmbientTalk. A problem with enabled-sets [TS89] is that they do not separate the synchronization concerns from the behavior. Indeed, the synchronization provided by the `enable` function and the conditions are crosscutting the methods `get` and `set` in the actor (table 6.17 on page 150). A more declarative synchronization mechanism is the use of *guards*, which were first introduced by Lucco [Luc87] and further studied by Löhr [Löh92] in the context of Eiffel. A guard associates a boolean expression to each method and will delay a message upon reception if the associated expression returns false.

Using guards, a bounded buffer object (shown in table 6.17 on page 150) can be synchronized with two guard statements. The implementations of these guards are expressed in the `syncMixin`.

```
syncMixin()::{
  init()::{
    .init();
    guard(put, { ptr<=MAX });
    guard(get, { ptr > 1 }) };

  requires(guardsMixin);
  capture()
}
```

```
guardsMixin()::{
  guards: vector.new();
  guardedMethods: vector.new();

  checkGuards()::{
    activated:
      vector.newWithVector(guardedMethods);
    guards.iterate({
      guardedMethod: el[1];
      guardCond    : el[2];
      if(not(guardCond()),
         activated.remove(guardedMethod))
    });
    enable(activated.asTable())
  };

  guard(method, condition())::{
    guards.add([method, condition]);
    if(not(guardedMethods
             .contains(method)),
       guardedMethods.add(method))
  };

  onProcessedMsg(msg)::{
    checkGuards()
  };

  init()::{
    .init();
    checkGuards();
    rcvbox.addAddObserver(
      thisActor()#onProcessedMsg)
  };

  requires(enabledSetMixin);
  capture()
};
```

Table 7.1: Language Mixin introducing Guards

Implementation in AmbientTalk

Table 7.1 shows the implementation of the `guardsMixin`, which has been implemented as an extension of the `enabledSetMixin`, shown in table 6.16 on page 150. The `guard` function takes two parameters: the first parameter is the method that needs to be guarded and the second parameter is a call-by-function condition for the guard that determines when the method can be activated. The `method` is paired with a `condition` closure in a table, which is added to the vector `guards`. Hence, the vector `guards` maintains a structure of all guards registered by an actor. `guardedMethods` maintains a structure of all guarded methods. The method `checkGuards` iterates[1] over all guard definitions and creates a vector containing the guards that returned false. Based on this vector the language construct `enable` is used to activate the methods whose guard evaluated to true. All guards are checked after processing a message, this is achieved by registering an observer for additions to the mailbox `rcvbox` in the `init` method.

Evaluation

Guards can be introduced both in a blocking and non-blocking concurrency model. Nevertheless, the resulting semantics is different. Consider that a method is invoked when the guard of that method returns false. In a blocking concurrency model it is the process of the calling active object that will block until the guard associated with the method returns true. This is in contrast to a non-blocking concurrency model where the process of the calling active object will not be blocked, irrespective of the result of the evaluation of the guard associated with that method. In that case the message will not be processed until

[1] The `iterate` method is defined with a call-by-function argument (discussed in section 5.3.2) that is parameterized with `el`. The expression provided as an argument is executed by the `iterate` method for each element in the mailbox, with `el` bound to that element.

the guard associated with that method returns true but the process context in which the method was invoked does not block. Note that this is also the case for the enabled-set synchronization mechanism, on which the guards mechanism is based.

A common evaluation criterion for synchronization schemes is its ability to deal with the inheritance anomaly problem [MY93]. The problem is that synchronization code cannot be effectively inherited without non-trivial redefinitions of the ancestors. In [MY93] it has been shown that traditional guards abstractions are insufficient to resolve the inheritance anomaly because the guards are immutably linked to methods. Our solution differs from this in that guards have been reflectively implemented and as such the guards configuration can be changed at run-time. This enables the developer to adapt the guards configuration based on the object extensions. It has been shown [MY90, MY93] that by manipulating the guards at the meta-level the effects of the inheritance anomaly can be resolved.

The implementation of this language construct is based on the reification of the communication traces. More particularly it is based on the reification of the stream of messages in the `inbox`. This might not be entirely obvious when looking at the code, since no mailboxes are accessed. However, the implementation of the `guardsMixin` is based on the implementation of the `enabledSetMixin` that has been implemented by moving messages from the `inbox` to the `delayedMbx` when they are to be delayed. This was discussed in section 6.5.3.

7.2.2 Token-passing continuations

Introduction

(One-way) asynchronous message passing in the actor model necessitates extensive use of callback methods to process replies to sent messages. This is illustrated by the example introduced in table 5.9 on page 115: when the method `get` is invoked, a callback `result` message is sent along to return the result to the caller. The use of callbacks complicates the structure of the code and is also known to be a source of race conditions [BY87].

Token-passing continuations are a programming abstraction, introduced in the actor language Salsa [VA01], to specify a message sending order based on the data flow between different actors. The data flow is specified by a sequence of message-based continuations parameterized by a *token*, which represents the return-value of the previous continuation. Its use is illustrated below:

```
checking<-getBalance() @
  savings<-transfer(token) @
    standardOutput<-println("transfer completed = ", token);
```

In the example, there are two actors representing accounts, `checking` and `savings`, and an actor `standardOutput` for printing. The example transfers the balance from the checking to the savings account and finally prints the status of the transaction. `<-` is used to denote an asynchronous method invocation. In Salsa, the @-sign defines a partial order on a sequence of messages. The last message will not be sent before the second message, and that one will not be sent before the first one. These messages represent the *continuations* of the computation, hence the name of the language construct. `token` is a special keyword within the scope defined by the previous token-passing continuation.

The `token` actually represents the return value of the previous message send. Therefore, the first occurrence of `token` refers to the result of `getBalance`, while the second occurrence refers to the result of `transfer`.

Implementation

In AmbientTalk the example above can be expressed using the `tokenPassing` language construct as follows:

```
tokenPassing(
  checking#getBalance(),
  savings#transfer(token),
  standardOutput#display("transfer completed = ", token, eoln)
)
```

Table 7.2 shows the implementation of the `tokenPassingMixin`, which adds the `tokenPassing` construct to allow imposing a dataflow order on actor messages. It is based on the design pattern to introduce the message-based scope, which has been discussed in section 6.6.2. The mixin introduces a special kind of `tokenMessage` extension for `message` objects. Such a message holds a continuation message, denoted with `cont`, and the `placeHolder`, which is used to represent unresolved tokens. We have chosen to represent this `placeHolder` as an actor for two reasons. First, an actor has a globally unique address, which cannot be forged. Second, representing the unresolved token as an actor enables the token to be used as a target address for token-based continuations of the form:

```
... @ token <- msg() @ ...
```

The token-passing continuation language construct is represented by the method `tokenPassing`, which takes a variable number of call-by-function arguments `exps`, parameterized with a formal parameter named `token`. Both *call-by-function* and *variable argument lists* were discussed in section 5.3.2. The body of this `tokenPassing` method reveals that two elements of the MOP are redefined. First, the `createMessage` method is redefined such that it creates a linked list of token messages with `first` pointing to the first continuation message. Second, the `send` method is overridden such that the active object does not send the messages evaluated in the context of the `tokenPassing` method. After redefining the MOP the method iterates over the expressions in the table `exps` and invoke them with the `placeHolder` as its argument in the dynamic scope. Hence, each expression in `exps` will be evaluated in a scope where the redefined MOP is visible such that all messages sent in this evaluation context will form a linked list of messages. Finally, the head of the linked list of token messages, referred to by `first`, is sent by performing a super-send to the `send` method.

The manner in which the token messages are handled is defined in the messages themselves, because we used the message scope design pattern introduced in section 6.6.2. The `process` method defined in `tokenMessage` calls `setToken` on the continuation message after the message has been processed. `setToken` replaces the `placeHolder` in the message with the result after which this continuation message is forwarded. The same meta-process is repeated until there are no more continuation messages to be sent.

```
tokenPassingMixin()::{
  placeHolder: actor(root.extend({ void }));

  tokenMessage(aMsg)::{ msg: aMsg.extend({
    cont       : void;
    placeHolder: void;

    isPlaceHolder(anActor)::
      anActor=placeHolder;

    setPlaceHolder(aPH)::placeHolder:=aPH;
    setContinuation(aMsg)::cont:=aMsg;
    getContinuation()::cont;
    setToken(aResult)::{
      if(isPlaceHolder(getTarget()),
         setTarget(aResult));
      args: getArgs();
      for(i: 1, and(not(is_void(args)),
                    i<=size(args)), i:=i+1,
        if(isPlaceHolder(args[i]),
           args[i]:=aResult))
    };

    process(behavior)::{
      value: .process(behavior);
      if(not(is_void(cont)), {
         cont.setToken(value));
         behavior.send(cont) });
      value
    }
  });
  msg.setPlaceHolder(placeHolder);
  msg };
```

```
  tokenPassing@exps(token)::{
    first  : void;
    msg: void;

    createMessage(s, t, n, a)::{
      ‘constructs a linked list of ‘
      ‘messages that were sent‘
      ‘with the continuation‘
      ‘pointing to the next message.‘
      if(is_void(first), {
         msg:=tokenMessage(
           .createMessage(s, t, n, a));
         first:=msg
       }, {
         next: tokenMessage(
           .createMessage(s, t, n, a));
         msg.setContinuation(next);
         msg:=next
       })
    };

    send(msg)::void;

    for(i:1, i<=size(exps), i:=i+1,
      dynamicScope(exps[i])(placeHolder));
    ‘add the first message in the outbox‘
    .send(first)
  };

  requires(messageScopeMixin);
  capture()
};
```

Table 7.2: Language Mixin for token-passing continuations

Figure 7.1 shows the messages that are sent as a result of executing the above expression. The diagram shows that it is the responsibility of the actors that processes the message to send the continuation message to the next target.

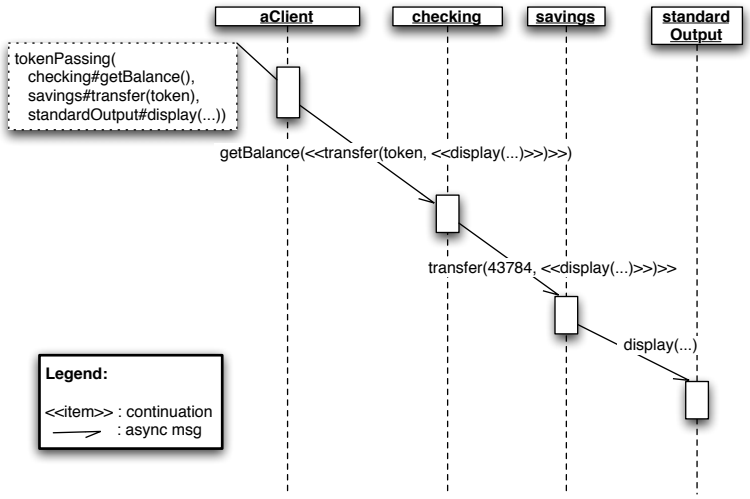

Figure 7.1: Behavior of Token-Passing Continuations

Evaluation

Token-passing continuations were first introduced in Salsa, which is based on the actor model and therefore respects the non-blocking communication characteristic. In fact, the token-passing continuations language construct was specifically designed to express coordination in the context of asynchronous communication and thereby enhance the expressiveness of this communication model.

Token-passing continuations have a limited expressiveness due to the rigid use of tokens. The main reason for this limited expressiveness is that the scope of a token is restricted to the previous continuation. For example, it is impossible to assign a token to some variable and reuse it in another context such as another message. If we want to achieve such flexibility we need to introduce a method that assigns the value represented by the token to the variable.

```
assignTokenToBalance(aBalance)::{
  balance:=aBalance;
  balance
};

transferMoney()::{
  tokenPassing(
    checking#getBalance(),
    thisActor()#assignTokenToBalance(token),
    checking#withdraw(token),
    savings#deposit(balance),
    standardOutput#display("transfer completed = ", token, eoln))
  )
}
```

The example above illustrates that such semantics forces us to introduce the method `assignTokenToBalance`, which is in fact another callback. Hence,

token-passing continuations do not completely eliminate the need for callbacks and can thus only be used to express a limited number of coordination patterns.

7.2.3 Futures

Introduction

A more flexible manner to express data flow-based distributed computations is to enrich the language with futures [Hal85, LS88] (also called promises). Futures are placeholders for the eventual result of an asynchronous call, similar to tokens but without the scope limitations found in token-passing continuations. A future is a proxy for the result to be computed. Once the result is computed the future is said to be *resolved*. A newly resolved future forwards the messages it received while it was unresolved to the result. Afterwards it forwards every incoming message immediately to the result. Futures allow programs to be written that use asynchronous communication but that still exhibit a control flow similar to that of code based on synchronous communication:

```
result: aQueue#pop();
...
result#print()
```

Futures are a well-established concept that has been incorporated in a variety of distributed programming languages [KB92, TMY94, Mil04, De 04]. Various flavors of futures exist. For instance, consider the case where the `print` message is sent to the future represented by the `result` variable before that future has been resolved. In some languages the process evaluating this expression is automatically and transparently suspended until the future has been resolved with the result of the `pop`. Other languages, such as Argus [LS88] and ABCL [YBS86], provide a construct to explicitly wait for the result to be computed. Such blocking future semantics is in conflict with the *non-blocking communication* characteristic of ambient-oriented programming languages that we advocated in chapter 3, because it is an implicit blocking receive statement. Hence, although one can write what appears to be sequential programs, these futures are in fact a potential source for (distributed) deadlocks.

The flavor of futures we integrated in AmbientTalk is based on the promises of the contemporary distributed language E [MTS05, Mil04], which were explicitly designed to support non-blocking communication. In E, when a message is sent to an unresolved promise, called a, then the process that sent the message is not blocked. Instead that message is queued at the unresolved promise a and a new promise b is associated with the enqueued message. This new promise b is returned as the result of the asynchronous method invocation sent to the promise a. This process, called promise pipelining [MTS05], is repeated until no more message are sent. At the moment promise a is resolved then its enqueued messages are automatically forwarded to the result represented by a. Eventually the promise b and other promises will become resolved and this process is repeated until no unresolved promises remain.

Quite often code may depend on the return value of an asynchronous invocation, despite the fact that this cannot be conveniently expressed by sending a message to the result. The following code excerpt gives such an example:

```
aFuture: aQueue#isEmpty();
...
if(aFuture, doSomething())
```

The conditional expression can only be evaluated after `aFuture` has been resolved. In concurrency models based on *wait-by-necessity* [Car89] this is achieved by blocking the evaluation of the conditional expression until `aFuture` has been resolved. An alternative approach is taken in E that introduces a `when` construct. This construct allows one to specify a closure that is scheduled for execution after a given future has been resolved. Note that the `when` construct does not block, it will execute its code block asynchronously when the future is resolved. The example below shows how this construct is used in AmbientTalk. The first argument of `when` refers to the future that must be resolved before the closure can be evaluated. The second argument refers to a block of expressions that are wrapped in a closure. In the scope of the closure the variable `content` refers to the result that was used to resolve `aFuture`.

```
aFuture: aQueue#isEmpty();
...
when(aFuture, if(content, doSomething()))
```

Implementation

The implementation of non-blocking futures is shown in table 7.3. An execution trace based on this implementation and the queue example from above with non-blocking futures is shown in figure 7.2. When a message is sent, a `future` actor is created and passed along as an attachment to the message. As described above the future acts as a placeholder for the result of the message. When the result is computed the future's `resolve` method is invoked. This method iterates over the `inbox` and forwards its messages to the result. After being resolved all new incoming messages are forwarded to the result by the observer method `onIncomingMsg`.

The `futuresMixin`, shown in table 7.4, adapts the message handling process using the `messageScopeMixin`. The method `createMessage` is redefined such that a new future actor `aFuture` is attached to every message sent. The message is extended using the `futureMessageMixin` shown below to be able to store the mail address of the `future` actor. The method `send` is redefined such that this future actor is returned as the result of the asynchronous call.

As shown below, the `process` method is refined in the `futureMessageMixin` of a message such that the value that results from running the method is sent to the future actor via `resolve` message.

```
futureMessageMixin(future)::{
  ...
  process(behavior)::{
    value: .process(behavior);
    future#resolve(value);
    value
  };

  capture();
}
```

The implementation of the `when` construct is included in `futuresMixin`, shown in table 7.4. The first argument of the `when` method is a *future* actor. The second argument is a call-by-function parameter (discussed in section 5.3.2) that adds the associated closure to a vector `whenBlocks`. This vector contains all the blocks registered as a result of invoking the `when` construct.

```
future:root.extend({
  resolved: void;
  subscribers: vector.new();

  new()::copy({
    subscribers:=vector.new();
    resolved:=void });

  init()::{
    inbox.addAddObserver(
      thisActor#onIncomingMsg) };

  subscribe(anActor)::{
    subscribers.add(anActor);
    if(is_resolved(resolved), {
       anActor#notify(resolved) })
  };

  resolve(content)::{
    subscribers.iterate(
      el#notify(content));
    resolved:=content;
    inbox.asVector().iterate({
      msg: el;
      forward(msg)
    })
  };
```

```
  forward(msg)::{
    if(and(is_actor(resolved),
           not(containsBehaviour(
                 msg.getName()))), {
      inbox.delete(msg);
      msg.setTarget(resolved);
      outbox.add(msg) })
  };

  onIncomingMsg(msg)::{
    if(not(containsBehaviour(msg.getName()))
       && is_resolved(resolved), {
      forward(msg)
    })
  }
});
```

Table 7.3: Implementation of Non-Blocking Futures in AmbientTalk

```
futuresMixin()::{
  whenBlocks: vector.new();
  newId     : 1;

  createMessage@args::{
    aFuture: actor(future.new());
    msg: .createMessage@args;
    msg.futureMessageMixin(aFuture)
  };

  send(msg)::{
    .send(msg);
    msg.getFuture()
  };
```

```
  invokeWhen(anId, content)::{
    whenBlocks.get(anId)@content
  };

  when(aFuture, code(content))::{
    whenBlocks.add(code);
    aFuture#subscribe(
      actor(futureObserver.new(newId,
                               thisActor())));
    newId:=newId+1;
    void
  };

  requires(messageScopeMixin);
  capture()
}
```

Table 7.4: Implementation of the `futuresMixin`

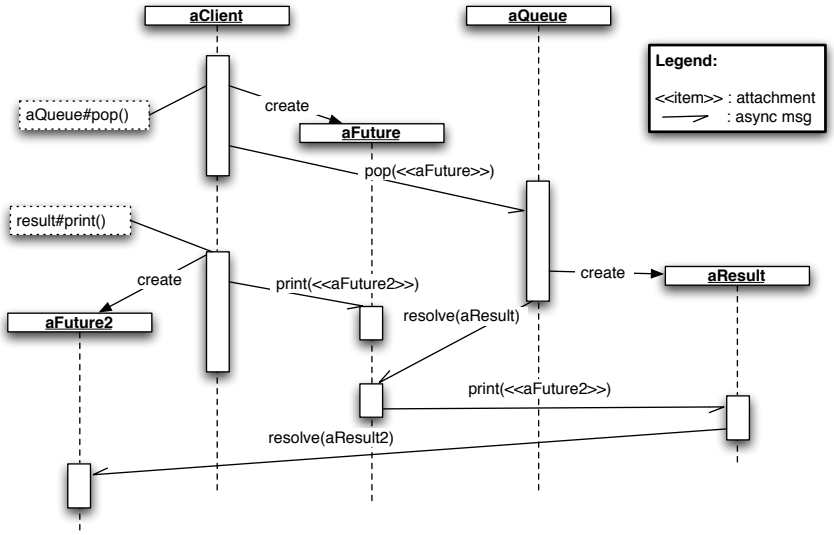

Figure 7.2: Behavior of Non-Blocking Futures

The `when` construct spawns a new `futureObserver` actor which is subscribed to the `future` actor. Below is the code that defines the behavior for the observer that is used to support the `when` construct:

```
futureObserver: root.extend({
  id: void;
  reference: void;
  new(anId, aReference)::copy{ id:=anId; reference:=aReference };
  notify(content)::reference#invokeWhen(id, [content])
});
```

The `futureObserver` actor is initialized with a unique number that corresponds to the index in the `whenBlocks` vector that needs to be executed and a reference to the actor that created the `futureObserver`. Upon notification, `invokeWhen` is sent which in turn looks up the closure and executes it with `content` bound to the resolved future value. Hence, the `when` construct is implemented by introducing the observer pattern in the future object shown in table 7.3 and the `futureObserver` actors are notified when the future receives a `resolve` message.

Evaluation

Non-blocking futures in combination with the `when` construct allow one to link an asynchronous message send to the code that is to be executed upon result propagation. The `when` construct thus aligns the computational context in which the message was sent with the one in which its result is handled without resorting to blocking semantics. Furthermore, AmbientTalk's non-blocking futures delay the delivery of received messages until the expected result is ready to receive them. Delayed messages are stored in the `inbox` of the actor that represents a future. This shows that reified communication traces are at the heart of realigning synchronisation with communication while strictly relying on non-blocking communication primitives as prescribed by the AmOP paradigm.

7.2.4 Combining Language Constructs

In this section we have introduced three language constructs, which were borrowed from existing concurrent and distributed object-oriented languages, to deal with coordination and synchronization of multiple processes in an expressive manner. Note that actors endowed with different combinations of these language constructs can coexist in the same system. For example, an actor based on token-passing continuations can interact with an actor based on the non-blocking futures. This flexibility originates from the message-based scope, discussed in section 6.6.2, such that the message itself and not the actor behavior defines how the message is processed.

It is even possible for the token-passing continuations to coexist with non-blocking futures in the same actor behavior. At first sight, it might not make sense to combine futures and token-passing continuations, since the semantics of token-passing continuations seems to be equally expressible using the futures language construct based on nested `when` constructs. The example used in section 7.2.2 can be rewritten as follows:

```
tokenPassing(
  checking#getBalance(),
  savings#transfer(token),
  standardOutput#display("transfer completed = ", token, eoln)
)

when(checking#getBalance(), {
  when(savings#transfer(content), {
    standardOutput#display("transfer completed = ", token, eoln) }) })
```

Both exhibit the same semantics, but their underlying messaging protocol is different. The protocol of the rewritten example is shown in figure 7.3 and can be compared to figure 7.1 on page 165[2]. It is clear that the version based on `when` is less efficient in terms of messages sent over the network. What is more, in the context of volatile connections the semantics is different. Suppose that the actors `aClient`, `checking`, `savings` and `standardOutput`, depicted in the figures, are running on separate devices, then the `when`-protocol requires that `aClient` must remain connected because it defines the "continuation" of the protocol, whereas in the protocol of the `tokenPassing` language construct the continuations are embedded in the message such that `aClient` does not have to remain connected after it sent the message.

The integration of token-passing continuations and non-blocking futures is not straightforward because both meta-mixin implementations of these language constructs define how a message is processed. Both language-mixins define a message processing protocol. However, the message processing protocol of token-passing continuations is only enabled in the context of the `tokenPassing` method due to the MOP that is only overridden in this method based on the local dynamic scope we discussed in section 6.6.2. Nevertheless, the message that is sent as a result of the token-passing continuation is to be combined with the non-blocking futures. Hence, the message processing protocols that implement the semantics of the token-passing continuations and non-blocking futures have to be combined. In the code, in tables 7.2 and 7.4, this composition of both protocols results from the mixin methods that are dynamically applied

[2]Note that figure 7.3 has been simplified by eliminating the `futureObserver` and `future` actors from the figure.

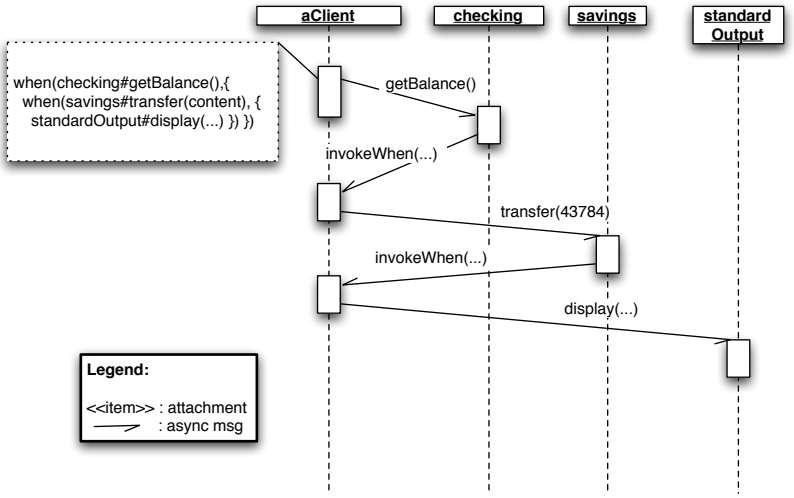

Figure 7.3: Token-Passing Continuations Expressed with nested `when`-statements

to the `message` objects in the `createMessage` method. A message sent in the context of both non-blocking futures and token-passing continuations will be a `tokenMessage` with a `futureMessage` as its parent that in turn has a `message` object as its parent. Each of these objects refines the `process` method of their parent such that processing the `tokenMessage` message will result in invoking the complete message processing protocol. Nevertheless, the fact that these language constructs can be combined results from their design and special care has to be taken during the design phase to make such combinations possible. Note that not all language constructs can be combined in such a manner. It might be that language constructs are inherently conflicting with one another such that their combination is impossible. Hence, the combination of language concepts and constructs has to be carefully evaluated per case.

7.2.5 Evaluation for AmOP

The language constructs presented in this section above show AmbientTalk's capabilities to express synchronization and coordination constraints in the face of non-blocking communication primitives. Many synchronization constraints are either expressed through the use of blocking statements such as in the case of wait-by-necessity [Car89]. In the language constructs above synchronization constraints are expressed in two different manners: First, in the case of guards and enabled-sets synchronization is achieved by removing messages that need to be synchronized from an `inbox` such that its scheduled execution is delayed. The use of mailbox observers allow one to intercept messages before they are processed without introducing a blocking communication primitive in the MOP. Second, in the case of token-passing continuations and futures the messages their execution is delayed because they reside in the `inbox` or `outbox` of an intermediary actor. The fact that in both these cases the synchronization is expressed in terms of mailboxes shows that the reified communication traces of

the AmOP paradigm lies at the hart of realigning synchronization constraints with the use of non-blocking communication primitives.

7.3 Ambient References

In this section we discuss programming language abstractions for *addressing* the services located on remote devices in the ambient. As a concrete example, consider a printer with a built-in printing service program. When the user declares that he wants to print a file from his PDA, and he is in close proximity to the printer, an appropriate service discovery algorithm should bring the PDAs objects in contact with the printing service. In regular object systems, acquaintance relations between objects are represented as object references (i.e. pointers). We therefore seek to explore abstractions for object references which can denote remote objects on a context-sensitive basis. The goal of such references is to both discover remote objects and to become a reference (i.e. a communication channel) to them. We name such object references *ambient references* [VDMD05].

Ambient references are a family of proxy abstractions, which enrich AmbientTalk's default service discovery mechanisms. These proxies define a suite of distributed reference abstractions that unify the two concepts *service discovery* and *communication* into a single concept. Ambient references can be thought of as active object references that sniff the ambient given a textual description. Once they discover actors fitting that description they become a communication channel to these actors. What is more, ambient references are resilient to the effects of volatile connections: upon disconnection ambient references try to rebind to a (potentially different) actor in the ambient fitting the description. An ambient reference is a reference to such a service in the ambient of a device. The use of ambient references is illustrated below:

```
printer: ambientRef("printer@300dpi");
printer#print(aFile)
```

`ambientRef` is parameterized with a pattern that denotes the service required from the ambient. Subsequently the message `print` is sent to the ambient reference. Note that the ambient reference need not be bound to an active object at the time a message is sent. In accordance with the non-blocking communication characteristic we discussed in section 3.3, the message sent will not block the sending actor. Which actor the `ambientRef` binds to, and how it handles the `print` message are dependent on the precise semantics used by the reference. The following section provides an overview of the various options.

7.3.1 Design Spaces

The behavior of an ambient reference is determined by a combination of design choices along three orthogonal dimensions:

Binding

An ambient reference can discover various suitable active object candidates to bind with in its ambient. Such a situation gives rise to two possibilities. First, it can bind to non-deterministically chosen active objects that are available in the

ambient. Such binding semantics has been studied by Black and Immel [BI93] in the context of replication. Alternatively, the ambient reference can semantically bind to all active objects available in the ambient. The ambient reference should then be considered a group communication abstraction. In this case we have to consider the multiple values returned by group invocations, which is further discussed in section 8.2.2. Ambient references with the former binding semantics are called *mono* ambient references, whereas ambient references with the latter type of binding semantics are called *multi* ambient references. Mono ambient references are useful in AmOP applications that require any ambient resource that fulfills a certain pattern. The printer scenario above illustrates such a use. Any printer that is able to print at 300 dpi can fulfill the printing job. On the other hand, multi ambient references can be used in AmOP applications that require communication with a group of ambient resources that match a pattern. For example, in an airport a digital broadcasting system might inform all passengers of a specific flight that boarding has started.

Re-Binding

When an unbound ambient reference discovers a suitable active object in its ambient it will bind to that active object. The sturdiness of this binding can vary. An ambient reference can bind to an active object and remain bound to that object forever. In that case, if the active object to which it is bound disappears from the ambient, then the ambient reference will rebind only when the active object to which it was first bound reappears in the ambient. Such a type of reference is called a *strong* ambient reference. This type of ambient reference is required by applications that require interactions with the exact same resource. For example, when a batch of `print` operations need to be performed by the same printer.

In contrast, the binding of an ambient reference can be *weak*. Upon disconnection a weak ambient reference will rebind with any other matching active object available in the ambient. Hence, it will rebind based on the naming information rather than the identity of the active object to which it was first bound. Weak ambient references are interesting for context-dependent interactions, where the identity of the active object the application communicates with depends on the ambient of the application. An example of this is a single `print` instruction that needs to be performed by any printer in the direct ambient of a device.

Delivery Guarantees

When a message is sent to an unbound ambient reference, the message can either be lost or its delivery can be guaranteed. The required semantics depends on the application. Guaranteeing eventual delivery of messages to ambient resources is not always useful and can be a waste of resources. For example, if an active object receives updated information at a high updating frequency then it does not matter if some messages are lost because the lost information becomes redundant due to the high frequency. In contrast, guaranteed eventual delivery of messages is useful in many applications because it enables abstraction over volatile connections.

7.3.2 Implementation

The different dimensions discussed above are reflected in the implementation of the `ambientRefBehavior` object, which is shown in table 7.5. This object reflects a weak multi ambient reference with guaranteed delivery. The `init` method registers the pattern that describes the actors to which the ambient reference will bind and subscribes a number of mailbox observers for additions on the `joinbox`, `disjoinbox` and `inbox`. These observers form the core implementation of the ambient references:

- `onJoined`: When first discovering an actor providing a matching pattern, all messages that were accumulated in the `inbox` are sent to that actor. This is a bootstrap phase such that all messages sent to the ambient reference while it was unbound are delivered to the first active object that is found in the ambient. Also note that each message is deleted from the `inbox` such that it doesn't get delivered multiple times. Furthermore, the reference is also added to the vector `refs` which maintains all the active objects in the ambient to which the ambient reference is bound.
- `onDisjoined`: If an active object that provides a pattern disappears from the ambient then all the pending messages to that active object are removed from the `outbox` and that active object's reference is removed from the vector `refs` such that the active object will no longer receive the messages sent to the ambient reference.
- `onIncomingMsg`: if the ambient reference receives a message and it is bound to at least one reference then the message is delivered to all the references to which it is bound. Finally, the message is deleted from the `inbox`.

Based on this implementation of weak multi ambient references we have defined a number of mixin methods to explore the other dimensions, shown in table 7.6. The `strongMixin` method overrides the `onJoined` and `onDisjoined` methods such that the vector `refs` remains invariable. As such all messages are delivered to the group of active objects independent from their current availability in the ambient. A *weak* ambient reference can be made *strong* by sending it a `snapshot` message. This method uses the `become` method to change the behavior of the ambient reference based on the `strongMixin` method. The `monoMixin` method restricts the binding of the ambient reference to a single active object in the ambient. The `noDeliveryGuaranteeMixin` removes messages from the `inbox` even though there is no binding available in the ambient. Based on these mixin methods we can create different variations of ambient references, as shown in tables 7.5 and 7.6.

7.3.3 Discussion

The implementation of the different types of ambient references shown above allows one to easily structure various ambient-oriented applications. This is because ambient references encapsulate two important aspects of ambient-oriented applications, *discovery* and *communication*, in a single abstraction.

There are many subtle choices that need to be made when implementing ambient references. For example, the delivery guarantee we have posed on the

```
ambientRefBehavior :: root.extend({
  pattern : void;
  refs    : vector.new();

  new(p) :: copy({ ... });

  init() :: {
    requiredbox.add(pattern);
    joinbox.addAddObserver(
      thisActor()#onJoined);
    disjoinbox.addAddObserver(
      thisActor()#onDisjoined);
    inbox.addAddObserver(
      thisActor()#onIncomingMsg)
  };

  onJoined(resolution) :: {
    ref: provider(resolution);
    refs.add(ref);
    toForward: inbox.asVector();
    toForward.iterate({
      if(not(containsBehaviour(
                el.getName())),
         { copiedMsg: el.copy();
           copiedMsg.setTarget(ref);
           outbox.add(copiedMsg);
           inbox.delete(el) })
    })
  };
  onDisjoined(resolution) :: {
    ref: provider(resolution);
    refs.remove(ref);
    outbound : outbox.asVector();
    outbound.iterate({
      msgTarget : el.getTarget();
      if (msgTarget ~ ref, {
          outbox.delete(el)
      })
    });
    disjoinbox.delete(resolution)
  };
```

```
  onIncomingMsg(msg) :: {
    if(not(refs.length()=0) &
       not(containsBehaviour(
             msg.getName())), {
       refs.iterate({
         copiedMsg: msg.copy();
         copiedMsg.setTarget(el);
         outbox.add(copiedMsg)
       });
       inbox.delete(msg)
    })
  };

  snapshot()::{ ... };

  strongMixin()::{ ... };

  monoMixin()::{ ... };

  noDeliveryGuaranteeMixin()::{ ... };
  capture()
});

WeakMonoAmbientRef(pattern) ::
  actor(ambientRefBehavior.new(pattern)
          .monoMixin());
WeakMultiAmbientRef(pattern) ::
  actor(ambientRefBehavior.new(pattern));
WeakMonoNoDeliveryAmbientRef(pattern) ::
  actor(ambientRefBehavior.new(pattern)
          .monoMixin()
          .noDeliveryGuaranteeMixin());
WeakMultiNoDeliveryAmbientRef(pattern) ::
  actor(ambientRefBehavior.new(pattern)
          .noDeliveryGuaranteeMixin());
```

Table 7.5: Implementation of Ambient References

```
ambientRefBehavior :: root.extend({
  ...
  new(p) :: copy({ ... });

  init() :: { ... };

  onJoined(resolution) :: { ... };
  onDisjoined(resolution) :: { ... };
  onIncomingMsg(msg) :: { ... };

  snapshot()::{
    become(this().strongMixin())
  };

  strongMixin()::{
    onJoined(resolution)::void;
    onDisjoined(resolution)::void;
    capture()
  };

  noDeliveryGuaranteeMixin()::{
    onIncomingMsg(msg) :: {
      .onIncomingMsg(msg);
      if(refs.length()=0,
         inbox.delete(msg))
    };
    capture()
  };
  ...
```

```
  ...
  monoMixin()::{
    onJoined(resolution) :: {
      if(refs.length()=0, {
        .onJoined(resolution) })
    };

    onDisjoined(resolution) :: {
      ref: refs.get(1);
      if(provider(resolution) ~ ref, {
        outbound : outbox.asVector();
        outbound.iterate({
          msgTarget : el.getTarget();
          if (msgTarget ~ ref, {
            inbox.add(el)
          })
        });
        .onDisjoined(resolution);

        if (joinbox.length() > 0, {
          next: joinbox.asVector().get(1);
          onJoined(next) })
      })
    };

    capture()
  }
});
```

Table 7.6: Implementation of alternative Ambient Reference Design Spaces

implementation of weak multi ambient reference ensures that a message will get delivered to at least one active object in the ambient. This delivery policy could be strengthened and an ambient reference could try to deliver all messages to all active objects that get bound to the ambient reference. Also, in the case of multiple remote actors to bind to an ambient reference could consider secondary criteria and base its binding choice on them, e.g. an ambient reference could look for any printer in the ambient and if multiple printers are found then the ambient reference binds to the printer with the highest resolution. These two cases show that the work regarding ambient references discussed above is not complete. Further experiments are needed to discover other useful dimensions and to properly evaluate the required semantics of the ambient references in different application scenarios.

Our notion of an ambient reference is very similar to that of a *handle* in the many-to-many invocations (M2MI) paradigm [KB02]. M2MI handles use Java interfaces to name other objects in a loosely coupled fashion and also employ asynchronous message passing. M2MI distinguishes between *unihandles*, *multihandles* and *omnihandles*. Roughly speaking, unihandles resemble strong monoambient references, while multi- and omnihandles resemble strong and weak multi-ambient references respectively. An omnihandle represents all objects in communication range implementing the handle's interface. A message sent to an omnihandle means "every object out there that implements this interface, call this method".

Although M2MI was of great influence to the design of our ambient references, there are some important differences. First, M2MI offers no delivery

guarantees: if a message is sent to an object which is not in communication range at that time, the message is lost. Hence, message sending and delivery are not decoupled as is the case with ambient references. The consequence is that the responsibility of guaranteed message delivery is passed on to the application itself. A second difference is that messages sent to M2MI handles do not return a value, requiring the use of callbacks as explained previously. On the other hand, ambient references can be combined with the futures language constructs we discussed in section 7.2.3 such that no callbacks are needed. Third, the construction of uni- and multihandles differs from the creation process of strong ambient references. In M2MI, objects must be explicitly attached to a handle, i.e. the set denoted by such a handle is explicitly enumerated. Strong ambient references dynamically discover their set content by taking a snapshot of a weak reference. In M2MI, there is no notion of such "snapshots".

7.3.4 Evaluation for AmOP

Ambient References establish and maintain a meaningful connection between two actors over a volatile connection. The implementation of ambient references heavily relies on AmbientTalk's reified environmental context. The reified environmental context plays a crucial role to manage the appearance and disappearance of communication partners. The pattern based lookup mechanism AmbientTalk inherited from the ambient actor model enables one to specify the type of service that is required from the ambient. Another important AmOP criterion that enables the implementation of ambient reference are the reified communication traces. The reification of these communication traces are used to flush messages accumulated during disconnection and to guarantee the delivery of messages. In the event of a disconnection weak ambient references retract sent messages that have not been delivered by moving them from the `outbox` into the `inbox` such that these messages. As a result, when a weak ambient reference senses another suitable object and binds to that object then these messages are delivered to the new object as opposed to the previously bound object. Hence, access to the `outbox` enables an active object to "unsent" undelivered messages.

7.4 Customized Message Delivery

The use of non-blocking communication primitives over volatile connections implies that the time a message is sent does not always coincide with the time that message is transmitted to the recipient. During this time interval the state of the application may change, due to the fact that devices are autonomous, such that it might no longer be necessary to transmit the message. An example of such a situation is a meeting scheduler which requests to schedule a meeting for tomorrow, while the message can't be transmitted until the day after tomorrow. In such situations the program needs to be notified such that it can take appropriate actions. For this reason we have introduced a new language construct named `due`. The situation sketched above can be captured as follows:

```
scheduleMeeting(anAgenda, meetingTime, meetingName)::{
  due(timeout(meetingTime - time() - ONE_HOUR), {
      anAgenda#schedule(meetingTime, meetingName)
  }, [ catch(true, thisActor()#recoverMeeting ) ])
```

```
};

recoverMeeting(catchedMsg)::{
  meetingName: catchedMsg.getArgs()[2];
  stdio#display("Meeting ", meetingName, " could not be scheduled timely.", eoln);
};
```

The example above shows two methods `scheduleMeeting` and `recoverMeeting` both implemented in an active object responsible for scheduling meetings. The `scheduleMeeting` method sends the message `schedule` to `anAgenda` in the scope of the `due` construct. The first argument of `due` is an arbitrary expiry condition that defines when a message is to be considered as obsolete. In this case the condition is a `timeout` set one hour before the meeting. The second argument is the expression evaluated in the context of the `due` language construct. The messages sent as a result of evaluating this expression are subject to the delivery semantics specified by the `due` construct. The last argument contains a table of `catch` statements. These `catch` clauses define how an expired message `msg` should be handled. Each `catch` statement consists of a condition and an action in the form of a first-class message. The first-class message associated with the first condition that returns true is sent upon expiration. In the example above there is only one `catch` statement with a condition that always returns true. The associated action `recoverMeeting` is parameterized with the message that expired. Furthermore, after a message expires it will no longer be delivered to its recipient.

7.4.1 Nested Due Blocks

It is possible to nest multiple `due` blocks such that multiple expiry-conditions can be applied. To illustrate the use we adapted the example above such that the meeting has multiple participants that are divided into two categories: required and optional participants. The former must be present at the meeting, whereas the latter are invited to the meeting but are not vital to the organization of the meeting. We require the `schedule` messages sent to the required attendees to be transmitted one week beforehand. Should one of these messages not be transmitted within this deadline then then a `cancelMeeting` message is sent to abort the meeting. An optional attendee that cannot be contacted is removed from the participants list by sending a `deleteFromParticipantsList`. These extra requirements are expressed as follows using nested `due` blocks:

```
scheduleMeeting(participants, optionalAttendees, meetingTime, meetingName)::{
  due(timeout(meetingTime - time() - ONE_WEEK), {
    due(timeout(meetingTime - time() - ONE_DAY), {
        participants.iterate(el#schedule(meetingTime, meetingName))
    }, [ catch(optionalAttendees.contains(msg.getTarget()),
               thisActor()#deleteFromParticipantsList ) ])
  }, [ catch(true, thisActor()#cancelMeeting ) ])
};
```

7.4.2 Implementation

As explained in section 5.2.3, AmbientTalk's default delivery policy guarantees eventual delivery of messages. Messages are stored indefinitely in the outbox of an actor until they can be delivered. The `due` language construct alters this policy by putting an expiration condition on outgoing messages. A `due`-block consists of an expiration condition (relative to the time at which a message is

sent), a 'body' closure and a table of `catch` statements that define the handler message to be sent upon expiration. When a message sent during the execution of the body expires, it is removed from the actor's outbox and the handler message is sent with the expired message as argument. The implementation of the `due` language construct consists of two separate language mixins:

- The **DueMixin** defines `due` which stamps all asynchronous messages sent while executing its body with an expiration deadline and a handler message to be sent upon expiration.
- The **ExpiryCheckMixin** makes an actor regularly check a specified mailbox in order to remove expired messages and to send their corresponding handler message.

The reason for separating the `dueMixin` and the `expiryCheckMixin` is that messages often get forwarded through different actors before reaching their destination. A typical example thereof is when actors are referred to indirectly via an ambient reference as explained in section 7.3: a message sent to an ambient reference first resides in the `inbox` of that ambient reference until the ambient reference can bind to a corresponding actor. After the ambient reference is bound to that actor the message is forwarded. Hence, a message may expire in the `inbox` of the intermediary ambient reference rather than in the `outbox` of the actor which originally sent the message. Another example is the future implementation, which was discussed in section 7.2.3. Such intermediary actors must therefore be able to detect expired messages even though they do not use the `due` construct. Hence, the `expiryCheckMixin` should be applicable to ambient references and futures.

The language mixin `dueMixin` is defined in table 7.7. The `dueMixin` installs the `due` construct in an actor and overrides the way its outgoing messages are created in order to stamp those messages by extending them with an attribute `sendTime` that contains the time the message was sent and a 'complaint message' which will determine how to react when the message expires. The overridden `createMessage` method first creates a message object `origmsg` by delegating to the default implementation. Subsequently, `origmsg` is extended with the slots provided that it was invoked in the dynamic context of a `due`-block (i.e. if `dueBlocks` contains the list of catch statements bound to `handlers` with their expiration condition `expiredCond` rather than `void`). To allow dynamic nesting of `due`-blocks, the current values of `expiredCond` and `handlers` are saved in a linked list and are restored upon returning from the `due` body closure. For each message that is sent in the `due`-block the appropriate handler is searched for with `findHandler`. This method returns the condition under which the message will be considered as expired and the associated expiry handler message. The `findhandler` method invocation is preceded and proceeded by a boolean that is flipped. This boolean is used to enable and disable the creation of expirable messages. This is necessary because in the context of the `findhandler` method handler messages are created encapsulated in the `catch` statements.

What remains to be explained is the `expiryCheckMixin`, shown in table 7.8, that registers a first-class message `tick` with a local clock actor which periodically sends this message. Upon notification, the actor examines messages in `aMailbox` stamped with a deadline to check whether they have expired. Ex-

```
dueMixin() :: {
  dueBlocks   : void;
  createDueMsg: true;
  FIRST       :: 1;
  NEXT        :: 2;

  createMessage(src, target, name, args) :: {
    origmsg:
      .createMessage(src, target, name, args);
    if(and(createDueMsg, !is_void(dueBlocks)),
        {
         createDueMsg:=false;
         msgHandler : findHandler(origmsg);
         createDueMsg:=true;
         if (!is_void(msgHandler), {
             extmsg: origmsg.dueMessageMixin();
             extmsg
               .setExpiryCond(msgHandler[1]);
             extmsg.setSendTime(time());
             complaintMsg : msgHandler[2];
             extmsg.setComplaintMessage(
               complaintMsg);
             extmsg
         },
         origmsg)
       },
       origmsg)
  };

  due(expiredCond(timeout), block(), catches)::
  {
    dueBlocks :=
      [ root.makeDueBlock(expiredCond,catches),
        dueBlocks ];
    block();
    dueBlocks := dueBlocks[NEXT]
  };

  catch(cond(msg), handler(msg))::
    root.makeCatchStatement(cond, handler);

  findHandler(msg) :: {
    findHandlerInDueBlock(handlers) :: {
      result : void;
      catchStatement : void;
      for(i: 1, (i <= size(handlers))
                  & is_void(result), i:=i+1,{
         catchStatement := handlers[i];
         if(catchStatement.condition(msg),
            { result :=
                catchStatement.handler(msg) })
      });
      result
    };

    currentDueBlock : dueBlocks;
    result : void;
    expiredCond: void;

    while(!is_void(currentDueBlock) &
           is_void(result), {
      result :=
        findHandlerInDueBlock(
          currentDueBlock[FIRST].localHandlers);
      expiredCond :=
        currentDueBlock[FIRST].expiredCondition;
      currentDueBlock := currentDueBlock[NEXT]
    });

    if(not(is_void(result)), {
      [ expiredCond, result ]
    }, void)
  };

  capture()
}
```

Table 7.7: Implementation of the dueMixin

```
expiryCheckMixin(aMailboxName, aDelay)::{
  myNotifier : ticker(aDelay);
  aMailbox: mailbox.get(aMailboxName)

  `invoked by myNotifier every aDelay millisec`
  tick()::{
    aMailbox.asVector().iterate(
      if(el.containsBehaviour("isExpired"),
         { if(el.isExpired(),
              { aMailbox.delete(el);
                complaintMsg :
                  el.getComplaintMessage()
                    .copy();
                complaintMsg.setArgs([ el ]);
                send(complaintMsg) })
         }))
  };
```

```
  init() :: {
    .init();
    rcvbox.addAddObserver(
      thisActor#onProcessedMsg);
    myNotifier#subscribe(
      thisActor()#tick)
  };

  onProcessedMsg(msg)::{
    .onProcessedMsg(msg);
    tick()
  };

  capture()
}
```

Table 7.8: Implementation of the expiryCheckMixin

pired messages are deleted from `aMailbox`, which is the mailbox associated to `aMailboxName`, and their handler message is sent to the appropriate actor.

7.4.3 Evaluation for AmOP

Due-blocks allow the sender to define, detect and deal with permanent disconnections. The due language construct shows that although AmbientTalk's default message delivery policy (discussed in section 6.4.5) implements a resumable communication model (where disconnections are not necessarily aligned with failures), one can still cope with permanent failures by reflecting upon an actor's communication traces: by having access to an actor's outgoing message queue which reifies its outgoing messages yet to be delivered, expired messages can be cancelled. Hence, the implementation of the `due` language construct was made possible due to AmbientTalk's support for reified communication traces.

7.5 Case Study: AmbientChat

In the previous sections we have discussed language features that deal with hardware phenomena we discussed in section 2.3. This section discusses the implementations of two similar chat applications that need to support these hardware phenomena. One of the applications is written in Java, whereas the other is written in AmbientTalk and uses some of the language features we introduced in the previous sections. These two applications allow us to analyze the differences between both languages.

Although a chat application is a fairly simple program it can be regarded as a "benchmark" for mobile distributed systems analogous to the use of the distributed whiteboard that is often used to compare how distributed platforms deal with recurring issues when modeling a distributed shared state. In the same way a chat application epitomizes many of the concerns that arise when developing an AmOP application:

- When two chat applications are in the ambient of one another they have to detect each other before they can communicate.
- Once the chat applications have detected one another they have to be able to exchange text messages.
- The chat application has to deal with volatile connections. i.e. communication failures resulting from volatile connections should not render the application unresponsive to other chat applications whose connection is not broken.
- Finally, the autonomy of a device requires that it can communicate with nearby devices without relying on any infrastructure.

In this section we discuss two such chat applications, called BlueChat and AmbientChat. The next subsection discusses the implementation of BlueChat. BlueChat was specifically designed to function on devices connected via a bluetooth interface. The implementation of the BlueChat application was published [Hui04] because many people find it difficult to build such applications using the current software technology. BlueChat is developed in Java on top of the J2ME platform. Java is considered to be the prototypical contemporary programming language that enables distributed programming for mobile devices. Other languages such as C#, which are based on the dot-NET platform together with the Compact Framework, exist but mostly have similar traits to the ones found in Java. Section 7.5.3 discusses the implementation of AmbientChat. The differences between these two applications are discussed in section 7.5.5, which analyzes both applications based on their lines of code and support for the hardware phenomena (discussed in section 2.3).

7.5.1 BlueChat

BlueChat [Hui04] is a chat application that was designed to function on small devices using Bluetooth and runs on J2ME compatible virtual machines. The application was written in Java and uses the Java API for Bluetooth Wireless Technology (JABWT for short) [Gro02]. The application forms a virtual chat room with other mobile devices that are in the communication range. Messages sent are received by all parties that are in the communication range. This is a small difference in functionality with the AmbientChat application we discussed below, which was designed to support point-to-point communication. Nevertheless, both applications must deal with the same hardware phenomena. Our goal is to discuss how BlueChat deals with these hardware phenomena. For this reason we first briefly discuss certain aspects of the implementation below (a more thorough can discussion can be found elsewhere [Hui04]).

BlueChat Implementation

The core functionality of the application is supported by five classes, whose implementation has been included in appendix B. Conceptually, BlueChat nodes in the network are represented as `EndPoint` objects. These objects enqueue `ChatPacket` objects that need to be communicated to other nodes in the network. `ChatPacket` objects encapsulate the low-level protocol of the chat application. Each `EndPoint` object transmits `ChatPacket` objects through the

support of `Reader` and `Sender` objects. Each of these objects is associated with its own thread and continuously reads and writes data from and to other `EndPoint` objects.

`NETLayer` is a singleton class that provides the core functionality of the chat application. Its main functions are to discover other devices in the communication range and provide an interface to facilitate the transmission of text in the chat. Once a BlueChat application is found then a connection is automatically established and an `EndPoint` object is created and maintained in a list in the `NETLayer`.

The NETLayer is linked with a graphical user interface based on the model-view-controller pattern. A message is emitted to the other BlueChat applications by invoking the method `sendString`. The graphical user interface is further updated through the notification of events such as users that enter and leave the communication range and messages that are received.

To give a taste of the BlueChat code we consider two aspects in more detail. First, we discuss how ambient resources are detected. Second, we discuss how communication is set up between devices. These two aspects are detailed below.

BlueChat Discovery

Other BlueChat applications in the ambient are detected through the use of classes related to discovery in the JABWT API. The discovery occurs in two phases. The first phase enables the discovery of bluetooth-enabled devices in the communication range. The second phase enables the discovery of services on these devices. The BlueChat discovery process is initiated in the `NETLayer.query()` method as shown below:

```
public void query()
{
  try {
    agent.startInquiry(DiscoveryAgent.GIAC, new Listener());
  }
  catch (BluetoothStateException e)
  { ... }
}
```

This code initiates the discovery of devices by launching an inquiry on a `DiscoveryAgent` object, whose implementation is part of the JABWT library. The first argument defines the discovery mode of this agent. `GIAC` refers to devices that are continuously discoverable. The second argument is a `Listener` object implemented as an inner class of the `NETLayer` class. This object provides an implementation for a `DiscoveryListener` interface. The implementations of the methods related to the first discovery phase are shown in table 7.9. The `deviceDiscovered` method gets invoked by the `DiscoveryAgent` for each device it discovers in the communication range. The newly discovered device is added to a list of pending endpoints. The endpoints are still pending because at this point in the discovery the application does not know if the device is running a BlueChat application. The method `inquiryCompleted` is invoked by the `DiscoveryAgent` when the inquiry finishes. The implementation of this method schedules the second discovery phase. `DoServiceDiscovery` is another inner class of `NETLayer`, shown in table 7.10. This class initiates a search for services through the same `DiscoveryAgent` for each pending endpoint. The second argument `UUID` refers to a unique identifier associated with

the bluechat application. The third argument refers to the device associated with the pending endpoint on which the service has to be searched and the last element refers to another `Listener` object that contains the callback methods that correspond to the actions that need to be performed when a BlueChat application service is detected. The implementations of these callbacks are shown in table 7.11. The method `servicesDiscovered` reveals what services, represented by `ServiceRecord` objects, run on the endpoint associated with `transId`. These `ServiceRecord` objects are placed in a map to avoid that a connection to the same endpoint is initiated twice. This issue is further discussed below. The method `serviceSearchCompleted` is invoked when the discovery agent has completed its search for services on a device. This method iterates over the map and opens the necessary connections with the BlueChat applications that were discovered. The streams associated with these connections are handed off to `Sender` and `Reader` objects to send and read data to and from the other BlueChat applications.

This discovery mechanism was implemented in an asynchronous fashion based on the observer pattern. Synchronization between callback methods is done based on a shared lock. A callback that depends on the actions of the other callback invokes the methods `wait` such that its process blocks until the other callbacks signals that it has finished its task using the `notifyAll` method. In this case the `DoServiceDiscovery` process has to wait until the `DiscoveryAgent` object has completed its search for services before it can remove all pending `EndPoint` objects.

Communication

We have discussed above that a connection is initiated for each newly discovered BlueChat service. These incoming connections are handled in the `NETLayer` class, which runs in a separate thread. The method that accepts these incoming connections is shown in table 7.12. The method starts by opening a server stream connector to listen for incoming connections. This connector is associated with the `UUID` that uniquely identifies the BlueChat application service that is embedded in its URL. After the connector has been created it is associated with a `ServiceRecord` object that provides low-level information about the service. This information was used above to facilitate the search for services on a device.

After this initialization round the thread goes into an infinite loop to listen on the connector for incoming connections. For each incoming connection the application checks if a connection to an endpoint already exists. It needs to do this because BlueChat applications are running on autonomous devices and it is possible that multiple BlueChat applications are simultaneously performing a connection with one another. Hence, due to concurrency that results two endpoints might be created to the same device. Nevertheless, this approach is not race condition free. Consider that immediately after the execution of the statement `endpt = findEndPointByRemoteDevice( rdev )` the discovery agent (who is running in a separate thread) performs a callback to `serviceSearchCompleted` (its implementation was discussed above). This callback invocation will also setup the connections. Hence, if the thread of control is switched back to the previous thread it will also setup connections to the same endpoint.

```
public class NETLayer implements Runnable  {

  ...

  class Listener implements DiscoveryListener
  {
    public void deviceDiscovered(RemoteDevice remoteDevice,
                                 DeviceClass deviceClass)
    {
        ...
        try
        {
          EndPoint endpt = new EndPoint(NetLayer.this, remoteDevice, null);
          pendingEndPoints.addElement( endpt );
        } catch (Exception e) { ... }
    }

    public void inquiryCompleted(int transId)
    {
      timer.schedule( new DoServiceDiscovery(), 100 );
    }

    public void servicesDiscovered(int transId, ServiceRecord[] svcRec)
    {
      ...
    }

    public void serviceSearchCompleted(int transID, int respCode)
    {
      ...
    }
  } // inner class Listener

  ...
} // NETLayer class
```

Table 7.9: Implementation of the Callback Methods associated with Device Discovery

```
public class NETLayer implements Runnable  {

  ...

  class DoServiceDiscovery extends TimerTask
  {
    public void run()
    {
      for (int i = 0; i < pendingEndPoints.size(); i++)
      {
        EndPoint endpt = (EndPoint) pendingEndPoints.elementAt(i);
        try {
          endpt.transId = agent.searchServices(null,
                                               new UUID[] { uuid },
                                               endpt.remoteDev,
                                               new Listener());
          synchronized( lock )
          {
            try {
              lock.wait();
            }
            catch (InterruptedException ex) {  }
          }
        }
        catch (BluetoothStateException e) { ... }
      } // for
      pendingEndPoints.removeAllElements();
      ChatMain.instance.gui_log( "", "You can start chatting now" );
    }
  }
}
```

Table 7.10: Inner class `DoServiceDiscovery` initiates the Discovery of Services

This race condition illustrates the difficulty that arises when programming for concurrent autonomous devices without a suitable concurrency strategy, even in the case of simple programs.

7.5.2 BlueChat Evaluation

In this section we briefly consider how the BlueChat application deals with the different hardware phenomena we discussed in section 2.3.

Ambient Resources

Ambient resources are detected through the use of a `DiscoveryAgent` object, whose implementation is part of the JABWT library. This agent enables applications to search for other bluetooth-enabled devices in its communication range. Once devices are found they can be searched for services. As can be seen in the code in tables 7.9 and 7.10 the reification of the environmental context requires the introduction of inner classes to deal with the events published by the agent. The code that deals with these events needs to manually maintain and establish the different connections with the chat applications. What is more, these events are asynchronously emitted and Java does not offer the right abstractions to deal with the coordination between these events. As a result coordination between events was implemented in an ad-hoc fashion based on the concurrency primitives offered by Java.

```
public class NETLayer implements Runnable  {

  ...

  class Listener implements DiscoveryListener
  {
    public void deviceDiscovered(RemoteDevice remoteDevice,
                                 DeviceClass deviceClass)
    {
        ...
    }

    public void inquiryCompleted(int transId)
    {
      ...
    }

    public void servicesDiscovered(int transId, ServiceRecord[] svcRec)
    {
      try {
        for ( int i=0; i< svcRec.length; i++ )
        {
          Util.printServiceRecord( svcRec[i] );
          EndPoint endpt = findEndPointByTransId( transId );
          serviceRecordToEndPoint.put( svcRec[i], endpt );
        }
      }
      catch (Exception e) { ... }
    }

    public void serviceSearchCompleted(int transID, int respCode)
    {
      ...
      for ( Enumeration records = serviceRecordToEndPoint.keys();
            records.hasMoreElements(); )
      {
        try {
        ServiceRecord rec = (ServiceRecord) records.nextElement();
        String url  = rec.getConnectionURL( ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false );
        StreamConnection con = (StreamConnection)Connector.open( url );
        EndPoint endpt = (EndPoint) serviceRecordToEndPoint.get( rec );
        if ( endpt != null )
        {
          endpt.con = con;
          Thread t1 = new Thread( endpt.sender );
          t1.start();
          Thread t2 = new Thread( endpt.reader );
          t2.start();
          endPoints.addElement( endpt );
          endpt.putString( NetLayer.SIGNAL_HANDSHAKE, localName );
        }
        } catch (Exception e)
        { ... }
      } // for
      serviceRecordToEndPoint.clear();
      synchronized( lock )
      {
        lock.notifyAll();
      }
    }
  } // inner class Listener

  ...
} // NETLayer class
```

Table 7.11: Implementation of the Callback Methods associated with Service Discovery

```
public class NETLayer implements Runnable {
  ...

  public void run()
  {
    StreamConnection c = null;
    try
    {
      server =  (StreamConnectionNotifier)Connector.open(
          "btspp://localhost:" + uuid.toString() +";name=BlueChatApp");

      ServiceRecord rec = localDevice.getRecord( server );
      rec.setAttributeValue( 0x0008, new DataElement( DataElement.U_INT_1, 0xFF ) );
      Util.printServiceRecord( rec );
      rec.setDeviceServiceClasses(
          SERVICE_TELEPHONY  );
    } catch (Exception e)
    {
      e.printStackTrace();
      log(e.getClass().getName()+" "+e.getMessage());
    }

    while( !done)
    {
      try {
        ChatMain.instance.gui_log( "", "Ready to accept connection. Wait..." );
        c = server.acceptAndOpen();

        RemoteDevice rdev = RemoteDevice.getRemoteDevice( c );
        EndPoint endpt = findEndPointByRemoteDevice( rdev );
        if ( endpt != null )
        {
        } else
        {
          endpt = new EndPoint( this, rdev, c);
          Thread t1 = new Thread( endpt.sender );
          t1.start();
          Thread t2 = new Thread( endpt.reader );
          t2.start();
          endPoints.addElement( endpt );
        }
      }
      catch (IOException e) {
        e.printStackTrace();
        log(e.getClass().getName()+" "+e.getMessage());
        if (c != null)
          try {
            c.close();
          }
          catch (IOException e2) {
            // ignore
          }
      }
      finally {
        // nothing to do here
      }
    } // while !done
  } // end run()
  ...
} // NETLayer
```

Table 7.12: `NETLayer.run` method accepts incoming connections

Volatile Connections

The implementation of the BlueChat application makes a distinction between graceful disconnections and unexpected disconnections. The former case occurs when the user intentionally exits the chat application. In this case a termination event is sent to other BlueChat applications which in turn clean up the necessary resources. In the latter case a BlueChat application unexpectedly becomes unavailable and the objects associated with the broken connection are removed. This case is captured through Java's exception handling mechanisms. BlueChat offers no other functionality to recover from failures.

Autonomy

The BlueChat application preserves the autonomous nature of the devices and does not rely on infrastructure. The autonomy is further preserved because the JABWT library offers a peer-to-peer discovery mechanism. As explained in section 3.3 the use of blocking communication primitives between shared components conflicts with the locking mechanisms. This application tries to avoid these problems by minimizing the use of locks such that resources remain available and the autonomy is preserved. Nevertheless, this strategy preserves the autonomy at the cost of race conditions. Race conditions can occur when multiple `Reader` objects concurrently update the user interface.

Natural Concurrency

The BlueChat application does not impose much consistency constraints. The concurrent nature of the BlueChat devices is preserved thanks to the use of callbacks and threads. However, the race conditions we have found in the code, of which one was described above, illustrates that even in simple applications the natural concurrency phenomenon can easily introduce bugs in code based on a thread based concurrency mechanism.

7.5.3 AmbientChat

AmbientChat is an instant messenger application designed to run on small mobile devices. The application's functionality is similar to BlueChat, but was written in AmbientTalk and illustrates the use of two language constructs we have discussed in this chapter, namely non-blocking futures (section 7.2.3) and ambient references (section 7.3).

The AmbientChat application consists of two components. An instant messenger component and an ambient sensor component. The complete code for the instant messenger behavior is shown in table 7.13. Each instant messenger has a `uid` that refers to the nickname of the chatter. `buddies` is a map of nicknames to other remote instant messenger components. `online` is a map of remote instant messenger addresses to nicknames. This map contains all current instant messenger components that are available in the ambient. `imSensor` refers to the ambient sensor component that is further discussed below. The `init` method initializes an ambient sensor to detect other instant messenger applications in the ambient. It also publishes two strings in the ambient. The `publish` method is implemented as `publish(aPattern)::{ providedbox.add(aPattern) }` and

relies on the mailboxes that reify the environmental context, which were discussed in section 4.5.4. `IMPATTERN` refers to a unique string that identifies applications as instant messenger components. Next to this pattern the component also publishes the nickname of the chatter in the ambient. The nickname is published to support device-independent communication. This feature of AmbientChat is further discussed below. Besides the initialization method there are a number of methods that make up the protocol of the instant messenger component. `getUID` returns the nickname of the chatter. `whoIsOnline` shows the nicknames of the chatters that are available in the ambient. `addBuddy` makes the instant messenger application aware of a chat buddy. This is achieved by adding a weak mono ambient reference to the buddy list[3]. `sendMessageTo` allows one to send a text message to a chat client by means of his nickname. In the case the chat client was previously unknown to a messenger he is automatically added to its buddy list. After this text is displayed on the device of the chat client to whom the text was sent it is also displayed on the instant messenger device that sent the text message. This is expressed by means of the `when` language feature discussed in section 7.2.3. The methods `onChatJoined` and `onChatLeft` make the user aware of other mobile devices that run an instant messenger that have become available or unavailable for communication, respectively. `IMActor` embeds the instant messenger passive object in an active object.

7.5.4 AmbientChat Evaluation

Now that we have described the AmbientChat application we can discuss how it deals with the different hardware phenomena we discussed in section 2.3.

Ambient Resources

Ambient resources are detected through an AmbientSensor component. The implementation of this component is shown in table 7.14. An ambient sensor component is parameterized with a pattern, which identifies the ambient resources it needs to reify, and two first-class messages that are sent when the ambient sensor detects an ambient resource matching the pattern has become available or unavailable, respectively. The ambient sensor component is implemented through the use of mailbox observers (discussed in section 5.4.4) on the `joinbox` and `disjoinbox` mailboxes.

Volatile Connections

In AmbientChat volatile connections are dealt with at three different levels of the application. Each level has its own specific purpose:

- First, when a connection breaks and is restored at a later stage then the communication is automatically resumed as a result of the default eventual delivery inherited from the underlying actor model. Hence, if a user decides to send a text to a buddy that is currently not in the

[3]Note that by changing this method such that a weak *multi* ambient reference is added instead of a weak *mono* ambient reference such that AmbientChat can send text messages to a group of chat clients. In that case a weak multi ambient reference denotes a virtual chat room and is initialized with the name of the chat room. Hence, in that case AmbientChat features group communication similar to BlueChat.

```
IMBehaviour :: root.extend({
  uid      : "anonymous";
  buddies  : void;
  online   : void;
  imSensor : void;

  new(id) :: copy({
    uid      := id;
    buddies  := smallmap.new();
    online   := smallmap.newWithComparator(key1~key2)
  });

  init() :: {
    imSensor := AmbientSensor(IMPATTERN, thisActor()#onChatJoined, thisActor()#onChatLeft);
    publish(IMPATTERN);
    publish(uid);
    stdio#display("Chat started as ",uid, eoln)
  };

  `Instant Messaging Protocol`
  getUID() :: { uid };
  whoIsOnline() :: { online.getValuesVector().iterate(stdio#display(el," ")) };
  addBuddy(buddyId) :: { buddies.put(buddyId, WeakMonoAmbientRef(buddyId)) };
  sendMessageTo(buddyId,text) :: {
     if (buddies.containsKey(buddyId),
         when( buddies.get(buddyId)#receive(uid,text), {
               receive(uid, text)
         }),
         { stdio#display(buddyId, " added to buddylist",eoln);
           addBuddy(buddyId);
           sendMessageTo(buddyId, text) })
  };
  receive(from, text) :: { stdio#display(from,": ",text,eoln) };

  onChatJoined(buddy) :: {
     stdio#display("InstantMessenger detected in the ambient: ",buddy,eoln);
     when (buddy#getUID(), {
        stdio#display("buddy online: ",content,eoln);
        online.put(buddy, content)
     })
  };

  onChatLeft(buddy) :: {
     if (online.containsKey(buddy),
         stdio#display("buddy offline: ",online.delete(buddy),eoln))
  }

}).futuresMixin();

IMActor(name) :: actor(IMBehaviour.new(name));
jessie: IMActor("Jessie");
jessie#sendMessageTo("Tom","Hi Tom");
jessie#sendMessageTo("Tom","How are you?");
```

Table 7.13: Instant Messenger Application in AmbientTalk

```
ambientSensorBehaviour :: root.extend({
  pattern : void;
  onJoinMsg : void;
  onDisjoinMsg : void;
  new(p,aJoinMsg,aDisjoinMsg) :: copy({
      pattern := p;
      onJoinMsg := aJoinMsg;
      onDisjoinMsg := aDisjoinMsg
  });

  init() :: {
      requiredbox.add(pattern);
      joinbox.addAddObserver(thisActor()#onJoined);
      disjoinbox.addAddObserver(thisActor()#onDisjoined)
  };

  onJoined(resolution) :: {
      copy: onJoinMsg.copy();
      copy.setArgs([ provider(resolution) ]);
      outbox.add(copy)
  };

  onDisjoined(resolution) :: {
      disjoinbox.delete(resolution);
      copy: onDisjoinMsg.copy();
      copy.setArgs([ provider(resolution) ]);
      outbox.add(copy)
  }
});

AmbientSensor(pattern,onJoin,onDisjoin) ::
  actor(ambientSensorBehaviour.new(pattern,onJoin,onDisjoin));
```

Table 7.14: Ambient Sensor

communication range of a device then that message will get automatically delivered at the moment a connection can be made. The user is aware when his text is actually transmitted because the text is only displayed on his instant messenger after it has been displayed on the recipient his instant messenger.

- The chat application supports offline communication *before* the users have made acquaintance. In other words, a user can send a text while he is offline to a buddy that he has never met before, based on his nickname. This is expressed through the use of weak mono ambient references. If a buddy is added to the buddy list then a weak mono ambient reference, which is parameterized with the nickname of the buddy, is added instead of a direct reference. Messages sent to the buddy are enqueued at the ambient reference until it detects an instant messenger application with a corresponding nickname. At that point the instant messenger automatically transfers all the enqueued messages.

  This scheme with the ambient references also makes communication device-independent. Hence, if a user decides to run his instant messenger application on another device (i.e. because he lost his previous device is broken or upgraded to a newer model) then the other instant messenger applications will be able to communicate with him when he configured his new device with the same nickname.

- Finally, if a chatter disappears from the ambient this is reported to the user through the ambient sensor component.

Autonomy

The autonomy of the device is preserved because the instant messenger application does not rely on any infrastructure to discover and communicate with peers. All shared objects interact only through non-blocking communication primitives. As a result no dependencies, originating from the use of locks, between actors are generated such that they remain entirely autonomous. Moreover, as opposed to the BlueChat application the autonomy does not come at the cost of race conditions.

Natural Concurrency

AmbientChat does not impose much consistency constraints, similar to BlueChat. Natural concurrency of the device is further preserved thanks to the non-blocking communication primitives. These primitives ensure that methods will always run to the end even in the face of communication failures. As such the instant messenger remains available to respond to other instant messenger applications or the user that is sending a message. Synchronization between AmbientChat applications is expressed based on the non-blocking futures and the `when` construct, discussed in section 7.2.3.

7.5.5 Discussion

Both the AmbientChat and BlueChat applications deal with the hardware phenomena we discussed in section 2.3. However, BlueChat's support to deal with

Type	AmbientChat	BlueChat	Factor
Application	22	45	2
Communication	5	210	42
Volatile Connections	3	80	26.7
Concurrency	5	37	7.4
Ambient Resources	43	102	2.4
Unspecified	10	111	11.1
Total	88	585	6.6

Table 7.15: Comparison of Lines of Code AmbientChat vs. BlueChat

volatile connections is limited compared to the support offered by the AmbientChat. What is more, AmbientChat is expressed in 88 lines of code as opposed to 585 lines for the BlueChat[4].

To further compare the differences between both applications we have colored each line of code in each application and attributed each color to a certain aspect that has be dealt with. The exact colored regions of the code can be found in appendices B and C. Table 7.15 summarizes the results of this identification of concerns. The most notable difference is the communication aspect. In the BlueChat application much of the code deals specifically with the communication between chat applications, whereas in AmbientChat very few lines of code deal explicitly with this aspect. One reason for this is that many of the communication aspects themselves are embedded in the AmbientTalk interpreter. What is more, the communication between objects has been aligned with the object paradigm such that communication is aligned with a simple asynchronous method invocation. Another noteworthy difference is the number of lines of code spent on dealing with volatile connections. In AmbientChat this is minimized thanks to the use of the ambient references and the fact that the communication is automatically resumed in the case of failures. In BlueChat volatile connections are more pervasive in the code because they are handled as exceptions.

7.5.6 Summary

In the second part of this chapter we have compared BlueChat and AmbientChat, two chat applications designed to run on mobile devices. Both applications are subject to the hardware phenomena, which we described in section 2.3, and need to deal with their consequences in the implementation. The former chat application was written in Java using the JABWT library whereas the latter was written in AmbientTalk. Even though the AmbientChat code was a factor of six smaller than the code of BlueChat it provides better support to deal with volatile connections. Such a comparison based on the number of lines of code of a single program is not a foolproof scientific way to compare two languages and draw definite conclusions. Nevertheless the number of lines of code can be regarded as a "litmus test" to measure the expressiveness of AmOP programs.

[4]To remove some of the bias of these result we have removed the documentation and only considered the code of the core functionality and the concurrent and distributed aspects.

7.6 Conclusion

This chapter has presented five tentative high-level AmOP language features: guards, token-passing continuations, non-blocking futures, ambient references and due-blocks. We have adhered to the (functional programming) tradition of modular interpreters to formulate these features as modular semantic building blocks – called language mixins – that enhance AmbientTalk's kernel. AmbientTalk's basic semantic building blocks (consisting of the eight first-class mailboxes, its mailbox observers and its reflective facilities) have been shown to be sufficient to implement these abstractions.

What's more, as discussed in chapter 4 the basic semantic building blocks make AmbientTalk adhere to the AmOP criteria. Hence, since these language mixins are constructed from these building blocks they adhere to the characteristics of the ambient-oriented paradigm. Table 7.16 summarizes the dependencies of the language constructs we defined and the communication characteristics of the paradigm. Synchronization and coordination constructs rely on the use of non-blocking communication and the reified communication traces. The combination of both allows one to express synchronization and coordination constraints without the introduction of a blocking operator. Ambient references heavily rely on AmbientTalk's support for the reified environmental context criterion in combination with the reified communication traces. Finally, long term disconnections can be addressed by changing AmbientTalk's default delivery policy. This delivery policy can be customized thanks to AmbientTalk's support for the reified communication traces criterion. The characteristic that objects should be classless is not included in this table, because this is a property that should be exhibited by the software components of a program as opposed to the language constructs.

The language constructs offer support to deal with the hardware phenomena we discussed in section 2.3:

- Volatile Connections
  All the language constructs presented in this chapter indirectly support volatile connections, because they rely on the fact that AmbientTalk's default communication strategy is to resume communication after a broken connection is restored. However, in section 7.4 we introduced the language constructs `due` that allows one to deviate from this default strategy, such that long-term disconnections can be dealt with appropriately.

- Ambient Resources
  Ambient references are a first wild proposal that deals with ambient resources which a device encounters in its immediate environment. They are promising abstractions, because they abstract both from discovery and communication at the same time such that they can also be used to deal with volatile connections through a means of intelligent rebinding of resources. As a resource is no longer available in the ambient of a device, it can search for a suitable replacement. Experiments have already shown that there is a need for different flavors of ambient references to deal with the different contexts in which they will be used. Further experiments are needed to uncover a taxonomy and make a selection of suitable semantics for AmOP applications.

kind of language construct:	name	non-blocking comm.	reified comm. traces	reified env. context
Coordination	guards	Yes	Yes	No
	tokens	Yes	Yes	No
	futures	Yes	Yes	No
Ambient References	Mono / Multi Weak	Yes	Yes	Yes
	Mono / Multi Strong	Yes	Yes	No
Long Term Disconnections	Due	Yes	Yes	No

Table 7.16: Evaluation of the Language Constructs

- Device Autonomy
  All language constructs presented in this chapter have been designed to communicate in a peer-to-peer fashion. Hence, the language constructs can be used to create applications that do not rely on infrastructure. What is more, all language constructs support the use of non-blocking communication primitives such that no dependencies are created as a result of the use of locks as discussed in section 3.3.

- Natural Concurrency
  Concurrency results from non-blocking communication between active objects. In section 7.2 we have discussed a number of abstractions to preserve consistency of concurrent access to resources.

In the last section of this chapter we validated some of the language constructs in the context of a chat application. This application replaces the whiteboard "benchmark" as the prototypical example in the context of mobile distributed systems. Indeed, the essence of communication between two chat applications consists of making the corresponding actors get acquainted and in handling the delivery, processing and result propagation of asynchronously sent messages between two autonomous actors that are separated by a volatile connection.

Surely, it is impossible to prove that AmbientTalk's building blocks are necessary and sufficient to cover all future AmOP features. Nevertheless, our analysis in chapter 3 strongly argues for their necessity and the expressiveness of our reflective extensions detailed in this chapter forms compelling evidence for their sufficiency. Thanks to the abstraction barriers offered by these reusable language constructs, our prototypical chat application counts merely 88 lines of AmbientTalk code. A chat application with similar goals called BlueChat implemented in Java using Bluetooth counts no less than 585 lines of code. BlueChat offers support for the discovery of ambient resources but has no provisions whatsoever to deal with temporarily lost connections.

Chapter 8

Advanced Experiments in Ambient-Oriented Programming

8.1 Introduction

This chapter continues the experimentation with language constructs for programming mobile distributed systems. However, the language constructs discussed in this chapter address two advanced themes important to construct specific types of mobile distributed applications.

First, in the next section we introduce language constructs that are designed to deal with group communication. Group communication is a useful abstraction for collaborative AmOP applications. The language mixins discussed in that section provide support for both intentionally and extensionally defined group references and introduces abstractions to coordinate the concurrency that is spawned from these results.

The second theme of this chapter is the employment of optimistic concurrency strategies. Optimistic concurrency strategies are important for mobile distributed systems because they provide advanced support for the autonomous nature of devices. We have already discussed in section 2.3 that autonomy implies natural concurrency. Optimistic concurrency strategies increase the autonomy of devices but do so at the cost of possible conflicts. In section 8.3 we study the concept of *virtual time*, which was first introduced by Jefferson [Jef85]. A distributed system based on virtual time maximizes concurrency based on an advanced distributed rollback mechanism, which is called Time Warp. Conflicts resulting from the concurrency are detected based on the link between time and causality. In section 8.4 we continue our study of optimistic concurrency strategies in the context of weak replication. We already noted in section 2.7.4 that weak replication increases the autonomy of devices because a service can be replicated onto a device such that the replicated service is local to the applications on that device. However, we noted that the current state of the art replicates data rather than objects. In this section we address this limitation and investigate weak replication of objects. We do so by combining Bayou's

anti-entropy protocol and the Time Warp algorithm. Finally, in section 8.5 we address the consequence of interactions with weakly replicated objects, namely the fact that objects have to deal with tentative data. Tentative data results from the interactions with replicated objects because weakly replicated objects are not necessarily fully synchronized. To deal with this aspect we extend the abstraction of non-blocking futures discussed in section 7.2.3.

8.2 Group Communication

Communication with groups of objects is important when considering cooperation in decentralized networks such as mobile networks. In the context of mobile networks group communication facilitates the discovery of groups of suitable active objects. The multi ambient references we discussed in the previous chapter are one such example. Furthermore, group communication enables decentralized propagation of events to active objects. For example collaborating distributed active objects that perform a particular task propagate such events such that each active object maintains an overview of the progress and can decide on its following actions.

The pairwise exchange of messages is not the best communication model to facilitate group communication. In AmbientTalk group communication can be facilitated by encapsulating the group members in what is called *multi references*. Such a multi reference allows addressing various actors with but a single message send. Members of the group can be declared in an *intentional* or *extensional* fashion. In the former case the groups are defined based on a common property. The multi ambient references of section 7.3 are one such example. They abstract over a group of active objects that provide the same pattern. In the latter case groups are defined by explicitly enumerating the members. Since the implementation of multi ambient references already provides us with an example of an intentional multi reference we first discuss the implementation of an *extensional* multi-reference. Next we address the issues that arise when considering the return values of a message sent to on a multi reference.

8.2.1 Extensional Group Communication

The multi reference is initialized with a table containing all actor addresses to be encapsulated by this reference, i.e. `group(a,b,c)`. The implementation of a multi-reference encapsulating an extensionally defined group of active objects is shown below:

```
groupBehavior::root.extend({
  group   : void;

  new(aTable)::copy(group:=vector.newWithTable(aTable));
  init()::inbox.addAddObserver(thisActor()#onIncomingMsg);
  onIncomingMsg(msg)::group.iterate({
    copiedMsg: msg.copy();
    copiedMsg.setTarget(el);
    outbox.add(copiedMsg) })
});

group@list::actor(groupBehaviour.new(list))
```

Upon receiving a message the group reference forwards all messages it receives to all encapsulated actor addresses. The `group` multi reference strongly

resembles a strong multi ambient reference, but the difference is that the former is created from an extensional enumeration of the members, whereas the strong multi ambient reference is created by taking a snapshot of an intentionally defined weak multi ambient reference.

8.2.2 Multi-Futures

An important aspect of group communication is to deal with the results returned by the different asynchronous invocations on multi references. In section 7.2.3 we have introduced *futures* as an abstraction to deal with results from asynchronous communication. We have seen that such a future is a proxy for the result returned by an asynchronous invocation. The logical extension of this abstraction is to conceive a future as a proxy for the group of values returned by group communications. Such a future is called a *multi-future* and may be used as illustrated below. The following code excerpt describes a personal shopping assistant that allows one to find the cheapest product in a shopping mall:

```
findCheapestProduct(shops, aProductName)::{
  products  : shops#snapshot()#getProduct(aProductName);
  prices: products#getPrice();
  bestPriceSoFar: void;
  bestProductSoFar: void;
  whenEach(prices, {
    thePrice: content;
    if(or(is_void(bestPriceSoFar), thePrice < bestPriceSoFar), {
      bestPriceSoFar:=thePrice;
      bestProductSoFar:=resolver;
      when(bestProductSoFar#getShopName(), {
        if(bestPriceSoFar = thePrice, {
          stdio#display("[partial result] Lowest Price: ", bestPriceSoFar, " at ",
                        content, eoln) ) }) }) });

   whenAll(prices, {
     when(bestProductSoFar#getShopName(), {
       stdio#display("[final result] Lowest Price: ", bestPrice, " at ",
                     content, eoln) }) })
}
```

The `shops` parameter refers to a weak multi ambient reference to all shops in the communication range of the shopping assistant. First, the shops in the communication range are queried for the product based on the name bound to `aProductName`. The result of this asynchronous invocation is a multi-future, which acts as a proxy for all the matching products. Subsequently, the prices of these products are requested from the products that were returned. This invocation returns another multi-future denoting the prices of the products. Upon this multi-future two new types of observers are placed: `whenEach` is similar to the `when` construct introduced above, but the code block is triggered *each time* a new result is returned. The code block is parameterized by `content` and `resolver`. The former is similar to the `content` parameter used in the `when` construct and refers to the new value that has been resolved. The latter refers to the active object that has produced the result. In the example, it is checked whether the new price bound to `content` is better than the best price that was encountered thus far. In that case the `resolver` parameter, which will refer to the active object representing the product with the best price, is requested for the name of the shop. The use of the `whenEach` construct allows the shopping assistant to print intermediate results even though not every shop has responded to the request. This is important in the context of active objects which may not

be available at the moment due to volatile connections. Note that the best price of the product is not only assigned to the `bestPriceSoFar` variable, but it is also stored in a local variable `thePrice`. This is necessary to prevent race conditions between the `whenEach` and the nested `when` invocation. Since `whenEach` relies on the result of the asynchronous invocation `getShopName` it is possible that a better price is found while waiting for the result of that invocation.

`whenAll` is the counterpart of `whenEach` and the code block is evaluated after the multi-future has been resolved with all the results. The code block is parameterized with `contents` and `resolvers`, two vectors containing all the computed values and all the resolvers at the corresponding indices. In the example above these parameters are not used because the final best price has already been computed and is assigned to `bestPriceSoFar` as a result of the `whenEach` statement. Note that `whenAll` can only be used with multi references that have a fixed number of members, such as strong multi ambient references and the group reference introduced above. For multi references where the number of participants can evolve over time, such as weak multi ambient references, it is impossible to determine whether a future has been fully resolved, because a member can be added to the multi reference at any point in time. Hence, when a participant is added to a multi reference after a future was determined to be fully resolved a conflict would arise. For this reason `whenAll` can only be used in combination with strong multi references that encapsulate a fixed group of participants such as strong ambient references and extensional groups. In the example this is addressed by sending the message `snapshot` to the weak multi ambient reference bound to `shops`, which turns it into a strong multi ambient reference as described in section 7.3. As a result this strong multi ambient reference will remain invariable with respect to the (un)availability of the references it points to in the ambient.

8.2.3 Implementation

Table 8.2 shows the implementation of multi-futures in AmbientTalk. The implementation of the multi-future is similar to the implementation of futures we discussed in section 7.2.3. The difference is that a vector containing the resolved values is maintained in `values`, as opposed to a single value in a future. Furthermore, the multi-future implementation also maintains a vector of `resolvers` that contains the active objects that resolved the multi-future with a value. It is important to determine when a future has been resolved with *all* results because only then can the `whenAll` blocks be executed. To determine whether a multi-future is resolved with all the results one needs to determine how many times it will be resolved in total. Note that this is not a trivial problem, because multiple values can be nested. Consider the following example:

```
mp3players: group(stores#getProduct("ipod"), stores#getProduct("zen"));
prices: mp3players#getPrice()
whenAll(prices, ...)
```

This example groups the results of two multi-futures resulting from the invocations `stores#getProduct("ipod")` and `stores#getProduct("zen")`. Determining when the multi-future bound to `prices` has been resolved with all the results of the `getPrice` invocation requires the coordination between the involved multi references (in this case `group` and `multifuture`). This coordination protocol is implemented based on the use of `whenAll` as can be seen in the

`forward` method in `multifuture`. For each message that is forwarded its old multi-future is stored in `oldFuture` and a clone of the message is created with a new multi-future attached. On each of these multi-futures a block of code is registered with `whenAll` such that after each of these attached multi-futures are resolved with all values the sum of all values is sent to `oldFuture` with a `complete` message.

Figure 8.1 shows the recursion of `complete` messages over multiple active objects. The scenario illustrates how the `complete` calls of `aFuture2` and `aFuture3` are accumulated by `group(a, b)`. This multi reference in turn notifies `aFuture1` that it is complete with two results.

The recursive implementation is similar for other multi references, such as `group` and strong multi ambient references. Note that their implementations shown previously do not include the recursion for didactic purposes. The recursion ends whenever a message is processed, which can be observed in the implementation of the `multiFutureMessageMixin` method shown in table 8.1. When an active object processes a message then the future associated with that message will represent a single value. Hence, when that message is processed the message `complete(1)` is sent to the multi-future.

The implementation of the `multiFuturesMixin`, which is the mixin that adds the `whenEach` and `whenAll` methods to an active object, has not been included because it is similar to the `futuresMixin` we have shown in table 7.4 on page 168. The implementations of the `whenEach` and `whenAll` methods are similar to the implementation of the `when` method. The only differences are that respectively a `whenEachSubscribe` or a `whenAllSubscribe` message is sent as opposed to the `subscribe` message sent in the `when` method. Finally, the `createMessage` method is changed such that the created message is extended with the `multiFutureMessageMixin` as opposed to the `futuresMessageMixin`, and a multi-future is assigned to the message rather than a regular future.

Similar to the implementation of regular non-blocking futures, this implementation of the multi-futures language construct utilizes the message-based scope (discussed in section 6.6.2). Hence, the `multiFuturesMixin` method has to be applied to the active objects that depend on the language construct and the `messageScopeMixin` method has to be applied to the receivers of asynchronous invocations that returned multi-futures. Both mixin methods can of course be applied to the same active object such that an active object can both utilize the language construct and be the receiver of such asynchronous invocations.

8.2.4 Discussion

The multi references and multi-futures discussed above introduce non-blocking group communication. The multi-reference implementations do not preserve the order of the messages that are forwarded. This design choice was influenced by the volatile connections characteristic. Consider that an active object is unavailable for communication, then if the order were to be preserved the subsequent messages that need to be forwarded to other members of a multi reference would be delayed until that active object would be available for communication. This design choice implies that `whenEach` blocks can be invoked out of order with respect to the order of the members of multi references. As a consequence, the order cannot be used to determine which active object has produced a certain result. For this reason a block is parameterized not only with the content of the

```
multiFutureMessageMixin(future)::{
  ...
  process(behavior)::{
    v: .process(behavior);
    future#resolve(v);
    this().getFuture()#complete(1);
    v
  };
  requires(futureMessageMixin);
  capture()
};
```

Table 8.1: Implementation of `multiFutureMessageMixin`

```
multifuture::root.extend({
  values                 : vector.new();
  resolvers              : vector.new();
  whenEachSubscribers    : vector.new();
  whenAllSubscribers     : vector.new();
  requiredResults        : void;
  sessions               :
    smallmap.newWithComparator(key1 ~ key2);

  new()::{ ... };

  notifyWhenAllSubscribers()::{
    if(hasReceivedAllResults(), {
      ... notify ...
    })
 };

  whenEachSubscribe(anActor)::{ ... };

  whenAllSubscribe(anActor)::{ ...  };

  hasReceivedAllResults():{
    and(not(is_void(requiredResults)),
        requiredResults = values.length())
  };

  complete(aTotal)::{
    requiredResults:= aTotal;
    notifyWhenAllSubscribers()
  };

  resolve(resolver, content)::{
    whenEachSubscribers.iterate({
       el#notify(resolver, content)});
    values.add(content);
    resolvers.add(resolver);
    notifyWhenAllSubscribers();
    inbox.asVector().iterate({
      msg: el;
      forward(content, msg)
    })
  };

  forward(anActor, msg)::{
    canForward: and(
      not(containsBehaviour(msg.getName())),
      is_actor(anActor))
    if(canForward, {
       copiedMsg: msg.clone();
       copiedMsg.setTarget(anActor);
       oldFuture: msg.getFuture();
       newFuture:
         actor(multifuture.new()
                  .futuresMixin());
       copiedMsg.setFuture(newFuture);
       whenEach(newFuture,
         oldFuture#resolve(resolver, content));
       whenAll(newFuture, {
         info : void;
         if(sessions.containsKey(msg),
            { info:= sessions.get(msg) },
            { info:= [0, 0] });
         info[TIMES_RESOLVED]:=
           info[TIMES_RESOLVED]+1;
         info[TOTAL]:=
           info[TOTAL] + contents.length();
         sessions.put(msg, info);
         if(and(hasReceivedAllResults(),
                info[TIMES_RESOLVED]
                  = requiredResults),
            { oldFuture#complete(info[TOTAL]);
              sessions.delete(msg) })
       });
       outbox.add(copiedMsg) })
    })
  };

  init()::{
    inbox.addAddObserver(
      thisActor()#onIncomingMsg)
  };

  onIncomingMsg(msg)::{
    values.iterate(forward(el, msg))
  }

});
```

Table 8.2: Implementation of Multi-Futures

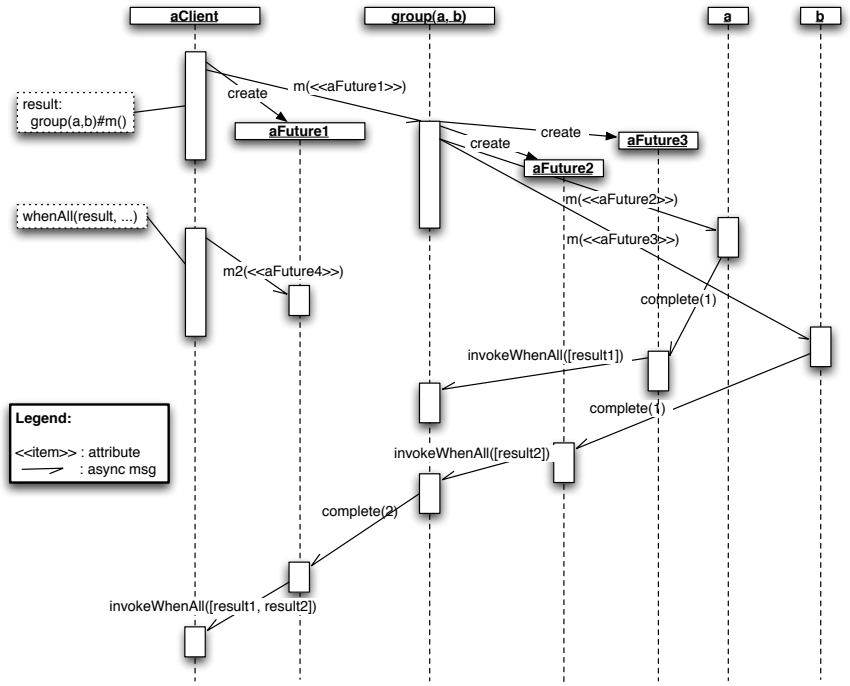

Figure 8.1: Determining when a multi-future has been completely resolved.

result, but also with the active object that resolved the multi future with the result.

8.2.5 Evaluation for AmOP

The `group` and multi-futures abstraction presented in this section have a an implementation similar to the ambient references and the futures presented in sections 7.3 and 7.2.3. The reified communication traces represented by AmbientTalk's mailboxes form a good abstraction to buffer incoming messages while the multi-future is not (fully) resolved. Once the multi-future becomes resolved with (partial) results these buffers are flushed so that the messages are forwarded to the results. Forwarding incoming messages is easily expressed thanks to AmbientTalk's mailbox observers. We have also demonstrated that these abstractions can be recursively implemented. In other words, the implementation of the coordination abstractions `whenEach` and `whenAll` are used to implement the behavior of `multifuture` objects.

8.3 Virtual Time

In the remainder of this chapter we investigate the use of optimistic concurrency control strategies in the context of mobile distributed systems. These concurrency control strategies are important because they best support the autonomous nature of devices and the natural concurrency that results from this

autonomy.

8.3.1 Introduction

In section 7.2 we have presented a number of synchronization constructs for describing concurrent interactions between active objects. All of these synchronization constructs are based on the notion of synchronized access to the state (only one message can be processed at a time) and delaying the execution of blocks of code until a certain condition is fulfilled. An alternative to these synchronization constructs are *reversible computations*. Reversible computations differ from traditional synchronization mechanisms in that they do not work on the basis of delaying the execution of code blocks, but instead rely on rollback mechanisms for maintaining a consistent state. Rollback mechanisms do not prevent inconsistent states, but instead thrive on the assumption that the parallelism will generally not lead to an inconsistent state. When an inconsistent state is detected then the computation is reversed to a previous consistent state.

Rollback mechanisms are best known from the context of database transactions where a series of SQL queries are ensured to atomically execute. Based on a rollback-retry cycle these SQL queries will be executed atomically exactly one time or not at all, which ensures a consistent state of the database. Argus [Lis92] transposed this notion of atomic execution of SQL queries to the level of objects by introducing atomic objects. Atomic objects allow concurrency within an atomic action and each such action can run atomic subactions in parallel. When an erroneous state is detected the transaction is aborted. Argus' notion of atomic actions are based on an elaborate locking mechanism that is difficult to uphold in the context of volatile connections as discussed in section 3.3.

A more generalized approach to these rollback mechanisms is Virtual Time [Jef85]. Virtual Time is a synchronization paradigm, which provides an alternative to delaying messages, based on reversible distributed computations. A Virtual Time system is a distributed system, whose nodes are executing in coordination with an imaginary virtual global clock that ticks virtual time. Each node has a local virtual clock that can be behind or ahead of the local virtual clock of other nodes in their distributed system. Virtual time systems have to adhere to two fundamental semantic rules:

1. The virtual send time of each message must be less than its virtual receive time.

2. The virtual time of each event in a process must be less than the virtual time of the next event at that process.

These two rules correspond to Lamport's clock conditions [Lam78] and ensure that if an event e_2 is a consequence of an event e_1 that the virtual time of e_1 precedes the virtual time of e_2. Hence, the arrow of causality will point towards future virtual time. Concurrency in Virtual Time results from the fact that two nodes can concurrently process messages irrespective of their local virtual clocks. Virtual Time is an optimistic concurrency approach because each individual local virtual clock of a node ticks time completely unsynchronized from the other nodes even though conceptually they are synchronized according to the two fundamental semantic rules. If a message arrives that should have been processed in the past at time rt (thus rt is strictly smaller than the virtual time

```
reversibleMessageMixin()::{
  receiveTime: void;
  sendTime   : void;
  id         : uid();
  negative   : false;
  setSendTime(aTime)::sendTime:=aTime;
  getSendTime()::sendTime;
  setReceiveTime(aTime)::
    receiveTime:=aTime;

  getReceiveTime()::receiveTime;
  getID()::id;
  setID(anId)::id:=anId;
  isPositive()::not(negative);
  isNegative()::negative;
  setNegative()::negative:=true;
  capture()
};
```

Table 8.3: Implementation of the reversibleMessageMixin

lvt of the node) then the state of the node is rolled back to the state at time *rt* and the message is processed. After the message has been processed at time *rt*, the messages whose effects have been rolled back are processed in the context of the new state of the node such that the node ends up again at time *lvt*. Note that rolling back entails not only that the local state changes between *rt* and *lvt* are undone but also the messages that are sent during this time period.

8.3.2 Implementation

Time Warp is the name of the distributed algorithm [Jef85] that realizes virtual time in a distributed system. In Time Warp each message has a virtual send and virtual receive time. The virtual send time of a message is set to the local virtual clock of the active object upon sending the message. The virtual receive time is the time at which the message is supposed to be processed at the target active object. The virtual receive time can be chosen by the sender, however it must abide by the first rule. Nodes sort the incoming messages based on their receive time and process the message with the smallest receive time first. If the message being processed has a receive time *rt* which is strictly smaller than the local virtual clock time *lvt* of the active object then the active object will rollback to the receive time of the message so that it can process the message at its intended receive time. Such a rollback of an active object not only entails that the previous state of the active object at time *rt* must be recovered, but also the communication performed by that active object between *rt* and *lvt* should be reversed. Reversing the communication is based on the notion of antimessages. For every message there exists an antimessage. The antimessage is exactly the same as its counterpart message except for its *sign*. This sign can be either positive or negative. Conceptually, sending a negative message to an active object will undo the effects caused by the positive message. The implementation of a Time Warp message is shown in table 8.3. The implementation adds the three attributes to a message we discussed above: `receiveTime` which is the virtual receive time of the message, `sendTime` which is the virtual send time of the message and `negative` is a boolean that indicates the sign of the message. Furthermore, another attribute `id` is added to a message to be able to uniquely identify each message in the system. Such a unique id is necessary to link messages to their antimessages because it is possible that two messages with exactly the same attributes are sent to a process.

Message Sign of Received msg	Anti-Message Found	Virtual Clock Condition	Action
-	nowhere	None	Do not process the msg Place msg in `rcvbox`
+	nowhere	msg.ReceiveTime >= local virtual clock	Process Message Place msg in `rcvbox`
+	nowhere	msg.ReceiveTime < local virtual clock	Reverse to local clock = msg.ReceiveTime Place msg in `inbox`
+	in `rcvbox`	None	Do not process the message Remove msg and antimsg from `rcvbox`
-	in `rcvbox`	None	Reverse effects of message Remove msg and antimsg from `rcvbox`
+ / -	in `inbox`	None	Remove msg and antimsg from `inbox`

Table 8.4: Decision Table for Processing Messages

Each active object is extended by the `reverseMixin` with a number of attributes to implement the rollback mechanism featured by Time Warp:

- An active object maintains a local virtual clock `clock` indicating the virtual receive time of the message that is currently being processed.
- A state queue `savedStates` maintains a list of `reverseBlocks` which contains checkpoint information of the active object's state to ensure that its state can be rolled back when a time conflict occurs.
- An input queue containing all messages delivered to the active object. The messages are stored in ascending order of the virtual receive time attached to each message by the sender.
- A received queue containing all messages that have been processed by the active objects in the past. These messages are also sorted in order of the virtual receive time attached to the message.
- An output queue containing the positive messages that are not yet transmitted to their target. These messages are ordered according to the message's virtual send time.
- A sent queue containing the negative copies of the messages that were communicated by the active object, also ordered according to the message's virtual send time.

These different message queues are represented in AmbientTalk as the mailboxes `inbox`, `rcvbox`, `outbox` and `sentbox` respectively.

Apart from these attributes the `reverseMixin` includes a number of methods to support the rollback mechanism:

- The `process` method, shown in table 8.5, implements a decision tree that incorporates the notion of antimessages in active objects. The rationale of this decision tree is that for each incoming message it checks if an antimessage is already present in the system. If this is the case then based on the current state of the active object and the message at hand a suitable rollback action is taken. The semantics of this decision tree is as follows:

- If no antimessage was found in the `inbox`, an antimessage is searched for in the `rcvbox`:
  - In the case that no antimessage was found in `rcvbox` there are two possibilities:
    - The message we are handling has a negative sign. This means that the negative message has arrived before the positive one. We do not process the negative message and store the message in the `rcvbox` in anticipation of the positive message that will arrive.
    - The message we are processing has a positive sign, in which case it can be processed if the virtual receive time of the message lies in the present or future of the virtual local clock. Before the message is further processed the local virtual clock is advanced to the virtual receive time of the message.

      If the virtual receive time of the message lies in the past according to the local virtual clock then a rollback is necessary to the virtual receive time of the message. Otherwise, the message is placed back in the `inbox` such that the message can be processed at its intended virtual receive time after the rollback has completed.
  - If such an antimessage is found in `rcvbox` there are two possibilities:
    - The antimessage that was found has a positive sign, which means that the message has been processed by the active object and as a result it will have caused side-effects in the form of state changes or messages that were sent to other processes. In this case the message is deleted from the `rcvbox` and a rollback is performed to the virtual receive time of that message to undo the effects.
    - Another possibility is that the antimessage that was found has a negative sign. In this case the negative message arrived before the positive one. Hence, since the positive one has not been processed the two messages annihilate one another.
- An antimessage was found in the `inbox`. In that is the case the two messages annihilate one another without any rollback. This is possible, because in this case the positive message is still in the `inbox` and therefore has not been processed and caused side-effects.

The semantics of this decision tree has been summarized in table 8.4.

- The methods related to the actual rollback mechanism are shown in table 8.6. The method `rollback`, which takes a virtual rollback time as an argument, will rollback the state of the active object to that virtual time in two steps. First, the active object retrieves the checkpoint state (with `getCheckpoint`) that was saved one time step before the virtual time to which the active object will rollback. Second, the rollback algorithm selects the messages in its `rcvbox`, that have a virtual receive time after the virtual time to which the active object has to rollback, and places them

back into the `inbox`. This will cause the messages that were received after the rollback time to be processed again once the state of the active object has been restored. The messages that have a virtual send time past the rollback time are removed from the `outbox`, thereby preventing the future communication of the active object from occurring.

The antimessages of the positive messages that have been transmitted are collected in AmbientTalk's `sentbox`. These negative messages are placed in the active object its `outbox` such that the effects induced by the active object's communication can be made undone. Finally, the saved state of the active object is restored. Restoring the state of an active object is comprised of restoring the `providedbox` and `requiredbox` mailboxes and the state of the passive object that defines the behavior of the active object. This last step can be conveniently done with the `become` primitive that the AmbientTalk kernel inherits from the actor model, discussed in section 5.4.1.

- Table 8.7 shows the changes that were made to the meta-interface. `onIncomingMsg` and `onSentMsg` are mailbox observers for additions on the `inbox` and `sentbox` respectively and keep their contents sorted by virtual receive and send time. The `createMessage` is overridden such that the `reversibleMessageMixin` is mixed into each message. The `send` method is overridden such that the virtual send time of the message is set to the local virtual clock of the active object. Furthermore, a virtual receive time is determined for each message. The actual setting of the virtual receive time is factored out in the method `determineReceiveTime`, because it is the setting of the virtual receive time that essentially determines how the Time Warp algorithm behaves. The higher the virtual receive time is set with respect to the local virtual clock, the less chance that a rollback will occur. As noted by Jefferson [Jef85] there is no single strategy that fits all applications.

8.3.3 Global Virtual Time

The method `checkpoint`, shown in table 8.6 is executed in the `process` method explained above and takes a deep copy of the passive object that defines the behavior of the active object. In the implementation a checkpoint is saved after executing each message. This raises a memory concern. Fortunately, it is not necessary to save a checkpoint after the execution of each message. Instead it is possible to save a checkpoint at certain virtual time intervals, because the rollback algorithm explained above will roll back to the nearest saved state before rollback time.

Another memory concern is that the `sentbox` and `rcvbox` seem to maintain a communication trace that goes back to the creation of the active object. Also, even though the number of checkpoints can be reduced by saving the state at certain time intervals, the queue of saved states only seems to grow over time. For this reason the Time Warp algorithm includes the notion of a global virtual time. The global virtual time is defined as the minimum of all local virtual clocks and the virtual send time of the messages that have been sent but not yet processed. This global virtual time is also known as the "commit

```
process(msg)::{
  if(and(isReversibleMsg(msg),
         ‘reverseIgnoreMsgs are reversible messages that are ignored‘
         ‘in the process and must not be reversed.  These include meta-‘
         ‘messages used to implement the Time Warp mechanism.‘
         !reverseIgnoreMsgs.detect(el.getName() = msg.getName())),
     { ‘find complementary message in the inbox‘
       idx: inbox.asVector().findFirst(
                              isReversibleMsg(el) &
                              (el.getID() = msg.getID()) &
                              (el.isNegative() = not(msg.isNegative())));
       antiMsg: if(is_void(idx), void, inbox.get(idx));
       if(is_void(antiMsg),
          { ‘no antimessage found in the inbox. Check the rcvbox.‘
            idx:=rcvbox.asVector().findFirst(
                                   isReversibleMsg(el) &
                                   (el.getID() = msg.getID()) &
                                   (el.isNegative() = not(msg.isNegative())));
            antiMsg:=if(is_void(idx), void, rcvbox.get(idx));
            if(is_void(antiMsg),
               { ‘no antimessage was found in the inbox and rcvbox‘
                 if(msg.isNegative(),
                    { ‘the negative message arrived before the positive one‘
                      ‘do not process it.‘
                      rcvbox.add(msg) },
                    { ‘check if we can process the message on time‘
                      if(msg.getReceiveTime() >= clock,
                         { ‘the message can be processed on time: ‘
                           ‘execute and advance time‘
                           clock:=msg.getReceiveTime();
                           v: .process(msg);
                           this().checkpoint();
                           v },
                         { ‘message arrives too late, rollback to its virtual ‘
                           ‘receive time‘
                           inbox.add(msg);
                           rollback(msg.getReceiveTime())  }) }) },
               { ‘an antimessage was found in the rcvbox‘
                 if(msg.isPositive(),
                    { ‘the message in the rcvbox is the negative message‘
                      ‘so we do not process the positive one‘
                       rcvbox.delete(antiMsg) },
                    { ‘the message in the rcvbox is the positive message‘
                      ‘Hence, we need to rollback‘
                       rcvbox.delete(antiMsg);
                       rollback(msg.getReceiveTime()) }) }) },
          { ‘an antimsg in the inbox was found‘
            inbox.delete(antiMsg) }) },
     { v: .process(msg);
       rcvbox.delete(msg); v }) };
```

Table 8.5: Implementation of the `process` method in the `reverseMixin`

```
rollback(aClock)::{
  if(aClock < clock, {
    cp: this().getCheckpoint(aClock);
    ‘move all processed messages
     after cp.clock from the rcvbox into
     the inbox‘
    rcvbox.asVector().iterate(
      if(el.getReceiveTime() > cp.clock,
         { rcvbox.delete(el);
           inbox.add(el) }));
    ‘remove the messages that have not
     been transmitted yet.‘
    outbox.asVector().iterate(
      if(el.getSendTime() > cp.clock,
         outbox.delete(el)));
    ‘move all antimessages after cp.clock
     from the sentbox to the outbox‘
    sentbox.asVector().iterate(
      if(isReversibleMsg(el) &
           (el.getSendTime() > cp.clock),
         { sentbox.delete(el);
           outbox.add(el) }));
    ‘sort the outbox such that the
     antimessages will be
     transmitted first‘
    outbox.sort(el.getSendTime());
    inbox.sort(el.getReceiveTime());
    ‘restore to the previous state‘
    restoreMailbox(provided,
                   cp.savedProvided);
    restoreMailbox(required,
                   cp.savedRequired);
    become(cp.getSavedState()) })
};
```

```
checkpoint()::{
  reverseInfo: root.makeReverseBlock(
    clock, this().deepCopy(),
    provided.asVector(),
    required.asVector());
  savedStates.add(reverseInfo))
};

‘selects the ReverseBlock that contains
 the rollback information for a rollback
 to aClock.‘
getCheckpoint(aClock)::{
  idx: savedStates.findFirst(
         el.clock > aClock);
  if(is_void(idx),
     idx:=savedStates.length());
  if(idx > 1, idx:=idx-1);
  savedStates.get(idx)
};

setCommitHorizon(aClock)::{
  committedClock:=aClock;
  savedStates.iterate(
    if(el.clock < (aClock-1),
       savedStates.remove(el)));
  sentbox.asVector().iterate(
    if(isReversibleMsg(el) &
       (el.getSendTime() < aClock),
       sentbox.delete(el)));
  rcvbox.asVector().iterate(
    if(el.getReceiveTime() < aClock,
       rcvbox.delete(el)))
};
```

Table 8.6: Implementation of the `rollback` method in the `reverseMixin`

```
reverseMixin()::{
  ...
  onIncomingMsg(msg)::{
    partialMailboxSort(
      msg, inbox, el.getReceiveTime())
  };

  onSentMsg(msg)::{
    if(isReversibleMsg(msg) &
       msg.isPositive() &
       !reverseIgnoreMsgs.detect(
          el.getName() = msg.getName()),
       'leave an antimessage in the
        sentbox'
       { msg.setNegative() },
       'sent negative messages leave
        no trace'
       { sentbox.delete(msg) });
    partialMailboxSort(
      msg, sentbox, el.getSendTime())
  };

  determineReceiveTime(msg)::{
    sendTicker := sendTicker+1;
    msg.setReceiveTime(
      clock+(sendTicker * step))
  };
  ...
```

```
  ...

  send(msg)::{
    if(isReversibleMsg(msg),
       { msg.setSendTime(clock);
         determineReceiveTime(msg) });
    .send(msg)
  };

  createMessage(s, t, name, s)::{
    msg: .createMessage(s, t, name, a);
    if(!reverseIgnoreMsgs.detect(
           el.getName() = name),
       msg.reversibleMessageMixin(),
       msg)
  };
  ...
}
```

Table 8.7: Implementation of the MOP in the `reverseMixin`

horizon", because it defines the virtual time that has become stable. In other words, no active object will request for a rollback to a virtual time before the global virtual time. Hence, the communication traces and the saved states that are associated with a virtual time before the global virtual time may be deleted. The global virtual time is regularly computed by a dedicated active object that consults all the active objects that have the `reverseMixin` applied. Afterwards, the active object informs all active objects of the "commit horizon" by invoking the `setCommitHorizon` on all these active objects. This method `setCommitHorizon`, which is shown in table 8.6, expunges all the communication traces and state queues before `aClock`.

8.3.4 Discussion

Concurrency in Time Warp results from the fact that even though the local virtual clocks of active objects are conceptually synchronized the active objects can concurrently execute messages. Two active objects in a Time Warp system can concurrently execute two messages irrespective of the state of their individual local virtual clocks under the condition that the two active objects have no causal relationship. If a causal relationship between active objects is exists and the corresponding actions conflict with respect to the two conditions of Lamport then the involved active objects are rolled back to a state where the actions no longer conflict and the system is resumed from that point on.

The algorithm that determines the global virtual clock requires that all the active objects in a system can be contacted in order to calculate the lowest

virtual clock. Hence, the concept of a global virtual clock does not scale in the context of long-lasting network partitions which can occur as a result of volatile connections in mobile distributed systems. Nevertheless, note that the other concepts in Time Warp, namely the rollback mechanisms based on local virtual time, are performed in a peer-to-peer fashion such that they can function properly in the absence of infrastructure. The only consequence of being unable to determine the global virtual clock is that it is impossible to determine the commit horizon raising the memory concern we discussed above.

8.3.5 Evaluation for AmOP

The Time Warp protocol was designed to function in the context of non-blocking communication. The design of the Time Warp protocol requires fully reified communication traces in order to implement its extended rollback functionality. Furthermore, the implementation of Time Warp naturally fits with the native mailboxes of AmbientTalk. Hence, an incoming message can be easily searched for its antimessage in the different mailboxes and based on the mailbox type the appropriate action can be taken. The implementation of this protocol also demonstrates the usefulness of AmbientTalk's `rcvbox` and `sentbox`. The `rcvbox` is useful to determine the scope of the concurrency conflicts. Thanks to these mailboxes it is easy to decide whether the effects of a message have to be undone or can simply be cancelled out by removing message and antimessage from the corresponding mailboxes.

As mentioned above the only concept in the Time Warp protocol that does not scale in the context of mobile distributed systems is the global virtual time. This concept is responsible for deciding on a commit horizon such that active objects can decide which messages can be expunged from the mailboxes. The process that computes the global virtual time requires access to all active objects involved in the Time Warp protocol such that the autonomy of devices is harmed. In the next section we combine Time Warp with another protocol such that the concept of a global virtual time is unnecessary.

8.4 Weak Replication

In this section we study the use of weak replication. Weak replication is an important technique for programming mobile distributed systems because it increases the autonomous nature of devices. The autonomy is increased because services can be replicated onto a mobile device such that the replicated service is local to the application and remains available, irrespective of the ability to communicate with other resources e.g. the device providing the original service. The problem with current weak replication strategies is that they replicate passive data as opposed to objects. In this section we show how to promote weak replication to the level of active objects.

8.4.1 Introduction

In section 2.7.4 we have discussed proposals to increase the data's availability in the face of disconnections based on weak replication of data. These proposals deal with raw data as opposed to objects. As a result the replication schemes are

concerned with replication of data rather than services. In this section we discuss the implementation of a language mixin to enable weak replication of services. When designing a replication protocol one can make a distinction between both active and passive replication [GS97]. With *active replication* each replicated service is sent the same message, as opposed to *passive replication* where only one designated replica handles each message and subsequently updates all other replicas. These updates usually convey the raw changes in the state of the replicated service. Hence, with active replication the services converge to the same state because they all process the same messages, while passive replication converges to the same state because the memory is updated by the replica that handled the message.

An important aspect of replication algorithms is the *linearization* [GS97] of updates, which consists of two properties:

- **Order** the updates that result from multiple client interactions should be handled by each replica in the same order.
- **Atomicity** if an update is applied to one replica the update must be applied to all replicas.

Linearization ensures that clients interacting with replicated services do not perceive inconsistent states. Linearization is important to ensure convergence of replicas because the rules state that replicated messages are applied in the same order at each replica. As a consequence, the changes to the state that result from processing these messages are applied in the same order such that the state of all replicas converges.

8.4.2 The Anti-Entropy Protocol

The linearization properties discussed above are important to fully understand the nature of weak replication algorithms. Weak replication means that replicas can be out of synchronization with one another. Nevertheless, clients interact with such replicas and, as a result, these clients can get information that is not up-to-date with respect to the other replicas. Such information is considered to be *tentative*.

The weak replica synchronization algorithm we discuss is based on the anti-entropy protocol [PST+97b], which was designed in the context of the Bayou project. The protocol was designed for weak replication of databases. Databases are synchronized as a consequence of applying (database) memory updates to replicas. In the synchronization protocol of Bayou a distinction is also made between tentative and committed updates. A tentative update is a provisional change made to the database that is to be verified in the future. Committed updates will result in the same state on all replicas and are final.

In a group of replicas, one replica is always designated as the master whereas the others are called slaves. When a client updates a slave replica this change is at first tentative. These tentative updates are continuously distributed to all other replicas that are available for communication and that have not yet received the update. When a tentative update is communicated from one slave to another slave it remains tentative. However, when a tentative update is communicated to the master replica then the update will be committed by the master replica. Each update that is committed by the master replica is assigned

```
anti-entropy(S,R) {
  Get R.V and R.CSN from receiving server R
  # first send all the committed writes that R does not know about
  IF R.CSN < S.CSN THEN
    w = first committed write that R does not know about
    WHILE (w) DO
      IF w.accept-stamp <= R.V(w.server-id) THEN
        # R has the write, but does not know it is committed
        SendCommitNotification(R, w.accept-stamp, w.server-id, w.CSN)
      ELSE
        SendWrite(R, w)
      END
      w = next committed write in S.write-log
    END
  END
  w = first tentative write
  # now send all the tentative writes
  WHILE (w) DO
    IF R.V(w.server-id) <w.accept-stamp THEN
      SendWrite(R, w)
    w = next write in S.write-log
  END
}
```

Table 8.8: Anti-Entropy Protocol

a monotonically increasing number, called a *commit sequence number* (CSN), which determines the order by which the slaves must commit that update. When a master replica synchronizes a slave replica it sends all the updates in the order they were committed. Although the master replica has the privilege to be the first to assign the CSN to a tentative update, it does not preclude slave replicas to distribute the committed updates to other slave replicas.

In the protocol each update to the database is called a *write* and these writes are used to update different replicas. Each write is tagged by three attributes: a write is assigned a monotonically increasing number, called an `accept-stamp`, by the replica that first received the write from a client. Furthermore, this write is also assigned the `server-id` of this replica. Hence, both the `accept-stamp` and the `server-id` uniquely identify each write in the system. Finally, the attribute `CSN` refers to the commit sequence number assigned by the master replica. If the write is still tentative then the `CSN` attribute is set to an infinite value. Table 8.8 shows the pseudocode (from [PST+97b]) of the anti-entropy protocol. The protocol describes how a replica `R` is updated by another replica `S`. The replica `S` first retrieves two pieces of information from the replica `R` that it will update:

- The version vector `R.V(X)` is a map that identifies the largest `accept-stamp` of any write known to `R` that was accepted from a client by a replica `X`.
- `R.CSN` is the `CSN` of the last committed write received by replica `R`.

The protocol consists of two parts: 1) sending the new committed updates from `S` to `R` and 2) sending the unknown tentative updates from `S` to `R`. The sending replica `S` can decide which tentative and committed writes are new to the receiving replica `R`, based on the version vector `R.V` and the commit sequence number `R.CSN`, respectively. This is done by means of simple number

comparison: i.e. if `R.CSN` is smaller than `S.CSN`, `S` knows that it holds some committed writes which `R` does not. The sending replica iterates over all the writes it knows, which are stored in what is called a *write-log*, and compares the `accept-stamp` of each write with the version vector `R.V` it received. If the committed write's `accept-stamp` is smaller than `R.V(w)` then `R` already has the write but has not yet committed it. In that case the complete write does not have to be transmitted; instead the `CSN` is sent along with the `accept-stamp` and `server-id`, which both uniquely identify the write, in a *commit notification* to the replica `R`. Otherwise the full write is sent to `R`.

An analogous check on the version vector is performed to update `R` with new tentative writes. This mechanism of sending each new committed and tentative message to the replica being updated, one by one, makes the protocol both incremental and peer-to-peer. It is incremental because an interrupted synchronization process can be resumed at any point in time without having to restart the synchronization from the beginning. In other words, it can be restarted based on the last update that was sent to the replica. It is peer-to-peer because any other replica can initiate and resume the synchronization with another replica, both for tentative writes as well as for the committed writes. However, the master replica is needed to promote a tentative write to a committed write for the very first time.

The master replica is a necessary ingredient to make all replicas consistent, because it imposes a total order on the committed updates. This order guarantees that committed updates fulfill the linearization property explained above. As a result committed updates are guaranteed to be applied in the same order for all replicas such that the effects of a committed update are guaranteed to converge the replicas to the same state. On the other hand, tentative updates are applied by replicas in the same order as they were applied by the replica that received them for the first time from the client. However, this is only a partial order with respect to all tentative writes applied by different replicas. To illustrate this, consider tentative writes t_R^i applied to different replicas, where R identifies the replica that first accepted the write and i determines the order in which it was accepted. Consider the following traces in replicas `X` and `Y`:

X: t_X^1 t_Y^1 t_X^2 t_X^3

Y: t_Y^1 t_X^1 t_X^2 t_X^3

Both execution traces are valid with respect to the partial order. In other words for all $t_R^i t_R^j$ for a particular R, $i < j$. As a consequence the resulting state of the replicas can differ from one another. Note that such a trace can occur in practice. For example, if `X` synchronizes `Y` after `X` has received t_X^1 and `Y` has received t_Y^1 from a client.

Committed writes have a globally unique commit sequence number that defines a total order by which they should be applied to a replica. However, to ensure convergence of replicas it is important that they apply the committed writes in the same order *on a consistent state*. Hence, committed writes must be applied *before* any tentative writes such that the committed writes are applied to the consistent state of the replica. As a result the replica will *rollback* all tentative writes to the database before it applies a committed write. Hence, a replica will always have all committed writes in the front of its write-log.

8.4.3 Experiment: A Unification of Anti-Entropy and Time Warp

In this experiment the Time Warp protocol and the Bayou anti-entropy protocol are combined such that messages as opposed to database writes can be replicated. The resulting system supports weak replication of objects rather than databases. The time warp protocol is useful in the context of the anti-entropy protocol, because it provides a mechanism to roll back the interactions between a group of objects. We have argued above that such a rollback is necessary in weak replicas to undo the effects of tentative updates such that a new committed update can be applied.

A unification of the time warp protocol with the anti-entropy protocol means that we have to find an appropriate mapping between the virtual receive time of a message and the ordering information found in Bayou writes. In Bayou there is an important distinction between tentative and committed writes. Tentative writes are not totally ordered, whereas committed writes are totally ordered. Their total order is determined by their commit sequence number. Therefore, in the case of committed writes we map the commit sequence number (CSN) onto the virtual receive time of messages. However, it is important that committed writes are executed before tentative writes to adhere to the linearization property. To ensure that tentative writes are processed after all committed writes the tentative writes are all assigned an *infinite* virtual receive time when they are sent. To ensure that the mutual partial order of the tentative writes themselves is maintained we relied on the anti-entropy protocol, which preserves this order because the tentative writes are transmitted in the order of the *write-log*.

Another issue to consider is the determination of the *commit horizon*. We have discussed in section 8.3.4 that a global virtual time does not scale in the context of network partitions that result from volatile connections. The commit sequence numbers found in Bayou can serve as a commit horizon, because a committed write is definite and is never subject to a rollback. Hence, the commit sequence number of the last write a replica has executed determines its commit horizon. As a result, the commit horizon found in each replica will make progress at its own pace depending on the committed writes that it receives from other replicas.

8.4.4 Interactions with Replicated Objects

The combination of Time Warp and Bayou's anti-entropy protocol also affects the client active objects that interact with the slave replica objects. A slave replica can interact with client objects by sending message to them. However, since the slave replica is possibly in a tentative state it is possible that the messages it sent become invalid in the future. For this reason, when a client active object is sent a message from a replica active object that is in a tentative state then the state of that client active object should also be considered as tentative from that point on. As a result it is necessary to consider the Time Warp protocol also in the context of client active objects. Hence, next to the virtual time in the replicas we also have to consider the local virtual clock of the client active objects. The local virtual clocks of active objects and replicas are influenced by one another due to the causal interactions with replicas. To understand the consequences of these causally related interactions it is important to consider

whether they result from messages that are sent in a committed or tentative state. In the remainder of this chapter we call messages that are sent in the context of an active object in a tentative state (thus when the message is sent the sender's local virtual clock time = ∞) *tentative messages* and messages that are sent in the context of an active object in a committed state (thus when the message is sent the sender's local virtual clock time < ∞) are called *committed messages*. Based on this distinction we have made a categorization of the different possible interactions. In this categorization the term "client active object" refers to both regular (not replicated) active objects and replica objects because both types of active objects can act as a client.

- A client active object sends a message to a slave replica.
  - If the state of the sender is committed:
    The message is processed tentatively (thus with its local virtual clock = ∞).
    Afterwards that message is replicated and will eventually be processed by the master replica after which the committed version of that message will be replicated back to all slave replicas. If the committed version of that message arrives back at a slave replica that previously processed this message tentatively then the Time Warp protocol will reverse the effects of that message. The reason for this is that the virtual receive time of that committed message will equal to the CSN (as explained above) and thus the virtual receive time of that message will be in the past for that slave replica because a CSN assigned by a replica is always smaller than infinity.
  - If the state of the sender is tentative:
    The message is processed by the slave replica, but it is not replicated. The reason for this is that since the sender's state is tentative it is unknown whether the message will actually be sent when the state of the sender gets committed. Replicating the tentative message would eventually lead to a commit of that message by the master replica. As the state of the sender gradually evolves to a committed state the tentative message in the replica will be reversed and sent again in the context of a committed state. At that point the message will be replicated such that it eventually gets committed.
- A client active object sends a message to a master replica.
  - If the state of the sender is committed:
    The message is processed and committed by the master replica and is replicated to all slave replicas when they synchronize.
  - If the state of the sender is tentative:
    The message is processed by the master replica, but it is neither committed nor replicated. The reason is the same as for such an interaction with a slave replica. The message is processed tentatively after all committed messages have been processed.
- A (master or slave) replica sends a message to a regular active object.
  - If the state of the sender is committed:
    The message is processed and committed by the receiver.

  - If the state of the sender is tentative:
    The message is processed by the receiver and its state is considered to be tentative from then on.

- A regular object sends a message to a regular active object.

  - If the state of the sender is committed:
    If the receiver is in a committed state it will process the message and commit it. Otherwise, the receiver performs a rollback to its committed state and processes the message in the context of its committed state. Afterwards the receiver processes the tentative messages (that were undone by the rollback) again and ends up back into its tentative state.
  - If the state of the sender is tentative:
    The message is processed by the receiver and its state is considered to be tentative from then on.

The rules above naturally fit into the Time Warp protocol if we consider that an active object is in a tentative state when its local virtual clock is *infinite*. If the local virtual clock is *infinite* and the active object sends a message then the virtual receive time of that message is set to *infinity*. If the local virtual clock of an active object is smaller than *infinity* then that active object is considered to be in a committed state. The virtual receive time of a message is not set in this case. Based on whether the receiver is a regular active object or a replica object it will be processed differently:

- The receiver is a regular active object and not a replica:
  In that case the message is processed based on the last *committed local virtual clock* of the receiver. Hence, if the local virtual clock of the receiver is infinite then conforming to the Time Warp protocol the clock is set to the past such that the effects of the tentative messages will be temporarily reversed and the message is processed in a committed context. Afterwards the tentative messages are automatically processed again.

- If the receiver is a replica the message will be executed with virtual receive time equal to the commit sequence number when it concerns a message that was committed by a master replica. Again, if the replica is in a tentative state when it receives such a message it will be rolled based on the Time Warp protocol. In the other case, that is when the message does not have a commit sequence number, it is tentatively executed with an infinite virtual receive.

Note that with these rules, the local virtual clock of a regular active object that is in a committed state will always indicate zero and will not increase over time. Hence, the commit horizon of regular active object is set to zero. On the other hand, when a replica is in a committed state its local virtual clock will be set to the commit sequence number of the last message it committed. Both regular active objects and replicas that are in a tentative state have their local virtual clock set to *infinity*.

8.4.5 On the Dynamics of the System

Active objects get "infected" with tentative messages as a result of direct or indirect interactions with replicas. In other words, the tentative messages spread throughout the system as a result of the causal relationships between active objects. The tentative messages in the system start to disappear as tentative messages become committed by the replicas. In the same way as active objects got "infected" with tentative messages they are now "disinfected" with committed messages. The dynamics of infection and disinfection of active objects with tentative messages results from the Time Warp algorithm. On the other hand the Bayou anti-entropy protocol brings chaos to order and is responsible for the fact that the global system moves towards a committed state.

8.4.6 Implementation

The `bayouMessageMixin`, shown in table 8.9, is used to extend messages with the necessary attributes to accommodate the anti-entropy protocol. The mixin overrides the method `getReceiveTime` of the `reversibleMessageMixin`, which was shown in table 8.3 on page 205, such that an infinite virtual receive time is returned when the message is tentative and the *commit sequence number* when the message is committed. The changes needed to incorporate the anti-entropy protocol in an active object are encapsulated in the language mixin called `replicaMixin`. Its overall structure is shown in table 8.10. This language mixin adds two custom mailboxes named `committed` and `tentative` to the active object that represents the replica. These two mailboxes represent the *write-log* of a replica. The `committed` mailbox contains the committed messages in the order of their *commit sequence number*, whereas the `tentative` mailbox contains the tentative messages executed by the replica in order they were received from other replicas and clients. Next to these two mailboxes each replica has an attribute `lastCSN`, which refers to the *commit sequence number* of the last committed message that was executed by the replica, `lastTS`, which refers to the last `acceptStamp` the replica assigned to a message it received from a client, and a vector `enabledReplicaMsgs`, which contains the list of messages that must be replicated upon synchronizing with another replica. The `replicaMixin` requires the `reverseMixin` explained in section 8.3. Depending on the role the replica active object has to play, it can be configured as a master or a slave with the `masterMixin` and `slaveMixin` respectively.

Table 8.11 shows the implementation of the `antiEntropy` method that synchronizes two replicas according to the pseudo-code shown in table 8.8. The methods that update the state of the replica that is being synchronized are shown in table 8.12. The message `commitNotification` is sent when the replica has already processed the tentative message that is to be committed. The message is searched for in the tentative mailbox based on its `acceptStamp` and replica server id. After the message has been identified it is removed from the `tentative` mailbox. Since the message was already processed when it was tentative it is also present in the `rcvbox` due to the underlying time warp protocol, discussed in section 8.3. The message is also removed from the `rcvbox` and is placed in the `inbox` with the newly assigned *commit sequence number*. As a result that message will be processed next, because the `inbox` is ordered on the virtual receive time. Due to the implementation of the `bayouMessageMixin` the

```
bayouMessageMixin()::{
  csn         : void;
  serverId    : void;
  acceptStamp    : void;

  setServerId(anId)::serverId:=anId;
  setAcceptStamp(anId)::
    acceptStamp:=anId;
  setCSN(aCSN)::csn:=aCSN;
  getCSN()::csn;
```

```
  getServerId()::serverId;
  getAcceptStamp()::acceptId;
  ‘if it is a tentative bayou ‘
  ‘message receive time = infinite‘
  ‘otherwise CSN‘
  getReceiveTime()::
     if(is_void(csn), infinite, csn);

  capture()
};
```

Table 8.9: Implementation of the `bayouMessageMixin`

```
replicaMixin()::{
  lastCSN   : 0;
  lastTS    : 0;
  versions  : smallmap.new();
  tentative : mailbox.new("tentative");
  committed : mailbox.new("committed");
  ‘enabledReplicaMsgs contains the list‘
  ‘of msgs are configured to be ‘
  ‘replicated.‘
  enabledReplicaMsgs  : vector.new();

  antiEntropy(aReplica)::{ ... };
  getReplicaStatus()::
    [lastCSN, versions];
  commitNotification(ts, aServerId, aCSN)::
    { ... };
  getVersion(aVersionVector, aServerId)::
    { ... };

  receiveCommitted(msg)::{ ... };
  receiveTentative(msg)::{ ... };
  processTentative(msg)::{ ... };
  processCommitted(msg)::{ ... };

  replicateMsg(aMsg)::{
    enabledReplicaMsgs.add(aMsg)
  };
```

```
  isBayouMsg(msg)::{
    msg.containsBehaviour("getCSN")
  };

  send(msg)::{ ... };

  masterMixin()::{
    process(msg)::{ ... };
    receiveTentative(msg)::{ ... };

    capture()
  };

  slaveMixin()::{
    process(msg)::{ ... };

    capture()
  };

  requires(reverseMixin);
  capture()
}
```

Table 8.10: Skeleton of the `replicaMixin`

```
antiEntropy(aReplica)::{
  when(aReplica#getReplicaStatus(), {
    theCSN     : content[1];
    theVersions: content[2];
    msg: void;
    while(theCSN < lastCSN, {
      msg:= committed.get(theCSN+1);
      if(msg.getAcceptStamp() <=
         getVersion(theVersions,
                    msg.getServerId()),
         { aReplica#commitNotification(
              msg.getAcceptStamp(),
              msg.getServerId(),
              msg.getCSN());
           theCSN:=theCSN+1
         },
         { committedMsg: msg.copy();
           aReplica#receiveCommitted(
             committedMsg);
           theCSN:=theCSN+1
         })
    });
    tentativeMsg: void;
    tentative.asVector().iterate({
      if(getVersion(theVersions,
                    el.getServerId()) <
         el.getAcceptStamp(),
         { tentativeMsg:= el.copy();
           aReplica#receiveTentative(
             tentativeMsg) }) })
  })
};
```

Table 8.11: Implementation of the `antiEntropy` method in `replicaMixin`

virtual receive time of the committed message will return the commit sequence number and the local virtual clock, which was previously infinite, will cause a rollback to occur. As a result the newly committed message will be processed in the order of its commit sequence number. The `receiveCommitted` method is based on the same mechanism and is called when committing a previously unknown message. There are two methods to process the tentative and committed messages. `processTentative`, which processes the former type of messages, places the message in the `tentative` mailbox and updates the version vector of the replica accordingly. `processCommitted`, which processes the latter type of messages adds the method to `committed` mailbox and updates the commit horizon. Both methods are called by the `masterMixin` and `slaveMixin` methods, shown in tables 8.13 and 8.14 respectively. In what follows, we discuss them both in detail.

The Master Mixin The `masterMixin` overrides the `process` method, which is part of AmbientTalk's MOP. This method first checks whether the message to be processed should be replicated. If this is not the case or the sender was in a tentative state (this is the case when the virtual receive time is infinite), the processing of the message is delegated in the else branch to the default Time Warp behavior. If the message is to be replicated then the the master replica first checks whether the message to be processed is a `bayouMessageMixin` if this is the case the master replica extends the message with this mixin and sets the attributes of the message accordingly. Subsequently, the message is processed via the `processCommitted` method discussed above.

A tentative message that is sent to the master replica by a slave will be received by master replica through the method `receiveTentative`. This method assigns a *commit sequence number* to the message and updates the version vector. Afterwards it is placed in the `inbox` as a result of the call to the `receiveCommitted` method. Due to the implementation of a `bayouMessageMixin` the message will also have that *commit sequence number* as its virtual receive time. Hence, if the master replica is in a tentative state (because it received a

```
commitNotification(
  anAcceptStamp, aServerId, aCSN)::
{ idx: tentative.asVector().findFirst(
         { (el.getAcceptStamp() =
            anAcceptStamp),
          & (el.getServerId() =
             aServerId) });
  msg: tentative.get(idx);
  tentative.delete(msg);
  idx:= rcvbox.asVector().findFirst(
         {  isBayouMsg(el)
            & (msg.getAcceptStamp() =
               el.getAcceptStamp())
            & (msg.getServerId() =
               el.getServerId()) });
  if(not(is_void(idx)),
     { msg:= rcvbox.get(idx);
       rcvbox.delete(msg);
       msg.setCSN(aCSN);
       inbox.add(msg) })
};
```

```
receiveCommitted(msg)::{
  inbox.add(msg)
};

processTentative(msg)::{
  tentative.add(msg);
  value: .process(msg);
  versions.put(msg.getServerId(),
               msg.getAcceptStamp());
  value
};

processCommitted(msg)::{
  committed.add(msg);
  value: .process(msg);
  this().setCommitHorizon(msg.getCSN());
  value
};
```

Table 8.12: Implementation of the different state-update methods in `replicaMixin`

```
masterMixin()::{
  process(msg)::{
    if(enabledReplicaMsgs.detect(
         el.getName() = msg.getName()) &
       'we do not replicate the message '
       'if it is tentative'
       (isReversibleMsg(msg) &
         (is_void(msg.getReceiveTime()) |
         (msg.getReceiveTime() < infinite))),
       { if(not(isBayouMsg(msg)),
            { msg:=msg.bayouMessageMixin();
              msg.setServerId(replicaId);
              msg.setAcceptStamp(
                lastTS:=lastTS+1);
              lastCSN:=lastCSN+1;
              msg.setCSN(lastCSN) });
```

```
          processCommitted(msg)
        }, {
          .process(msg)
        })
  };

  receiveTentative(msg)::{
    lastCSN:=lastCSN+1;
    msg.setCSN(lastCSN);
    versions.put(msg.getServerId(),
                 msg.getAcceptStamp());
    receiveCommitted(msg)
  };

  capture()
};
```

Table 8.13: Implementation of the `masterMixin` method in `replicaMixin`

```
slaveMixin()::{
  process(msg)::{
    if(enabledReplicaMsgs.detect(
         el.getName() = msg.getName()) &
       ‘we do not replicate the message‘
       ‘if it is tentative‘
       ((isReversibleMsg(msg) &
          (is_void(msg.getReceiveTime()) |
          (msg.getReceiveTime() < infinite))) |
        ‘bayou msg is always replicated‘
        isBayouMsg(msg)),
       { if(not(isBayouMsg(msg)),
            { msg:=msg.bayouMessageMixin();
              msg.setServerId(replicaId);
              msg.setAcceptStamp(lastTS:=lastTS+1) });
         if(is_void(msg.getCSN()),
            { this().processTentative(msg) },
            { lastCSN:=msg.getCSN();
              this().processCommitted(msg) })
```

```
       }, {
         .process(msg)
       })
  };

  receiveTentative(msg)::{
    inbox.add(msg)
  };

  capture()
};
```

Table 8.14: Implementation of the `slaveMixin` method in `replicaMixin`

message from an active object sent in the context of a tentative state) it will have an infinite local virtual time and the underlying time warp protocol will initiate a rollback.

The Slave Mixin The `slaveMixin` also overrides the `process` method and performs the same checks as the `masterMixin`. However, the implementation differs from the `masterMixin` in that it will make a distinction between tentative and committed messages, whereas the master replica will commit independently of the type of message.

A tentative message that is received by the slave replica is placed in the `inbox`. Its virtual receive time will be infinite and it will be executed after all other messages that have been received with an infinite virtual receive time. This together with the order in which the anti-entropy method sends the tentative messages ensures that the tentative messages will be executed in the correct order.

8.4.7 Discussion

The system resulting from the combination of Bayou's anti-entropy protocol and the Time Warp protocol cannot be considered as a traditional virtual time system, because the two semantic rules set forth by Jefferson [Jef85], which we discussed in section 8.3, are not necessarily adhered to. More particularly, this is the case for the first rule that states that the virtual send time of a message must be less than its virtual receive time. There are two instances in which this rule can be broken:

1. The first is when the virtual receive time of a message is infinite. This is the case where the state of the sender is tentative. Hence, the virtual send time equals the virtual receive time.

2. The second is when no virtual receive time is set, as discussed above. In this case the virtual receive time depends on the receiver of the message and its state. Consider that a regular active object, which is in a committed state, sends a message to the master replica which is also in a committed state. According to the rules and their realization, both discussed in section 8.4.4, the virtual receive time will not be set and the message will be handled in the committed context of the local virtual clock of the master replica. The local virtual clock of the master replica is set to the commit sequence number of the last message it processed. Hence, the local virtual time of the master replica is possibly greater than the virtual send time of the message, which is zero in the case of a regular committed active object.

As a result of these violated rules the arrow of causality will not necessarily point in the direction of future virtual time. Note that with a different set of rules it might be possible to create a mapping that fulfills these two semantic rules. For example, by dividing the virtual time into a lower band for committed messages and a higher band for tentative messages. Within these bands one could assign virtual receive times that adhere to the semantic rules. Nevertheless, we have chosen not to take this approach because it could lead to unnecessary rollbacks. These rollbacks would be caused by the fact that the sender, which generally has to decide on a virtual receive time, cannot make any estimation about an appropriate receive time for a message due to the volatile connections.

8.4.8 Evaluation for AmOP

In this section we have demonstrated AmbientTalk's support for custom mailboxes. The implementation of Bayou's anti-entropy protocol was facilitated by the introduction of a `tentative` and `committed` mailbox to respectively store the tentative and final communication traces of active objects. What is more, the implementation of the protocol is conceived as an extension of the `reverseMixin` we discussed in the previous section. The necessary rollback semantics, which are required for the implementation of the anti-entropy protocol, are expressed by moving messages from one mailbox to another. Reversing the effects of a message is simply expressed by manipulating mailboxes. The graceful composition of both the `reverseMixin` and the `replicaMixin` demonstrates AmbientTalk's expressive mixin composition mechanism.

This language construct enables *weak* replication of active objects. Weakly replicated active objects are available for client active objects even when the master replica is not available (i.e. due to volatile connections). The rollbacks that result from tentative messages that are sporadically committed over time occur automatically. What is more, the changes that result from these rollbacks are automatically and sporadically propagated over the mobile network as explained above. Hence, this language construct provides a high-level abstraction to increase the autonomy of devices even in the face of long-term disconnections. However, the use of weakly replicated objects also introduces tentative data in the applications which has to be dealt with explicitly.

8.5 Support for Tentative Data

8.5.1 Introduction

The distinction between tentative and committed data that results from weak replication plays an important role in the way clients have to deal with the information that results from interactions with weak replicas. For example, tentative data could be shown in a special color in the user interface such that users know that the information is not final. Likewise, certain irreversible actions (typically interactions with the real world) should not be executed by the application based on tentative data. Hence, the distinction between tentative and committed data sometimes ought to be made explicit for the developer. For this reason we have extended the non-blocking futures of section 7.2.3 to support distinguishing between tentative and committed data. Suppose an agenda application contains weakly replicated calendars of a user's contacts and the user wants to schedule a meeting with one of them. After the meeting has been scheduled a reservation for a room for this meeting has to be made.

```
meetingName: "Meeting about AmbientTalk";
when(wolfgangCalendar#reserve(tomorrow(), 1400, meetingName), {
  if(content,
    { roomManager#reserve(tomorrow(), 1400, meetingName);
      stdio#display(meetingName, " has been scheduled.", eoln) },
    { stdio#display(meetingName, " has NOT been scheduled.", eoln) })
});
```

Suppose that the calendar of Wolfgang is a weak replica that runs on the device that schedules the meeting. The invocation `wolfgangCalendar#reserve(...)` will first resolve the future with a tentative value because the invocation is performed on a slave replica representing Wolfgang's calendar. Suppose that reservations for the room manager are irreversible, then it is necessary to postpone this reservation until Wolfgang is guaranteed to be available for the meeting. We have therefore extended the `when`-construct of section 7.2.3 such that it also indicates whether the block of code is executed in the context of tentative or committed results. This distinction is made with the variable `isTentative` that is available in the scope of the block passed to `when`. Another change in the semantics of the `when` construct is that the block can now be executed multiple times. The block is executed each time the future is resolved with a new value. The same future can be resolved multiple times as a result of the rollback mechanism we described in the previous section. It is only after the future has been resolved with a committed result that the block will never be executed again. Based on this extension we are able to adapt the example code as follows:

```
meetingName: "Meeting about AmbientTalk";
when(wolfgangCalendar#reserve(tomorrow(), 1400, meetingName), {
  status: if(isTentative, "[tentative] ", "[committed] ");
  if(content,
    { if(not(isTentative), roomManager#reserve(tomorrow(), 1400, meetingName));
      stdio#display(status, meetingName, " has been scheduled.", eoln) },
    { stdio#display(status, meetingName, " has NOT been scheduled.", eoln) })
});
```

This adapted version of the calendar example checks whether the value bound to `content` is committed and only then is the room reserved. Another change in the example is that the output is now prefixed with a tag that indicates whether the data shown on the screen is final or can be changed in the future.

Next to the extended `when`-construct we have also added a variation called `whenReverse`. This construct has the same semantics of the extended `when`-construct, but it takes a third argument that is evaluated before the future is resolved again with another value. Based on this mechanism we can further specialize the rollback procedure featured by the Time Warp protocol.

```
meetingName: "Meeting about AmbientTalk";
whenReverse(wolfCalendar#reserve(tomorrow(), 1400, meetingName), {
  `same as above`
  ...
}, {
  `warn user of change`
  stdio#display("Warning: status changed!")
});
```

The two extensions based on the `when`-construct allow one to deal with the tentative data that arises from the use of weakly replicated objects. Note that this tentative data arises from the autonomous nature of the devices, which was discussed in section 2.3. Indeed, the autonomy of the devices is supported because applications can interact with replicas that run on the same device. Since the replica is local to the application it is always available to the application such that it does not depend on the availability of another device to have access to the replicated service. The replicas themselves also have a high degree of autonomy thanks to their weak synchronization protocol. However, replicas must be able to synchronize from time to time in order to commit messages.

8.5.2 Implementation

Table 8.15 shows the extension of the implementation of non-blocking futures we introduced in section 7.2.3. The tentative future implementation adds an extra attribute `isTentative` that indicates whether the future was resolved with a result in a tentative or committed context. The method `resolve` is overridden such that it takes an extra parameter `isTentativeResolve` that sets the `isTentative` attribute. The future observers are also notified along with the context in which the future was resolved. The future observer, shown in table 8.16, notifies the active object to invoke the closure subscribed by the `when`-construct. However, before doing so we first send an antimessage for the previous notification such that it is undone by the active object. Note that the `createMessage` method is overridden in order for the virtual receive time of `invokeWhen` to be set according to the rules we discussed in section 8.4.4.

The `tentativeFuturesMixin`, shown in table 8.17, extends the `futuresMixin`. It overrides the `when` and `invokeWhen` methods to deal with the tentative or committed evaluation context. The `when` language construct's call-by-function argument `code` has an additional parameter `isTentative` that indicates the context in which the future was resolved. `whenReverve` adds a call-by-function argument `reverse` that is evaluated in `invokeWhen` if the future was previously resolved with another value.

8.5.3 Evaluation for AmOP

By integrating support for distinguishing between tentative and committed results into the non-blocking futures of section 7.2.3, we provide a basic mechanism to deal with conflicts arising from volatile connections and natural concurrency.

```
future:=future.extend({
  isTentative: void;

  subscribe(anActor)::{
    subscribers.add(anActor);
    if(not(is_void(resolved)) &
       not(isTentative),
       { anActor#notify(resolved,
                        isTentative) })
  };
```

```
  resolve(content, isTentativeResolve)::{
    isTentative:=isTentativeResolve;
    resolved:=content;
    subscribers.iterate(
      el#notify(content, isTentative));
    inbox.asVector().iterate({
      msg: el;
      forward(msg)
    })
  };

  capture()
});
```

Table 8.15: Implementation of the extended future

```
futureObserver:= root.extend({
  id: void;
  reference: void;
  createMessage(s, t, n, a)::{
    msg: .createMessage(s, t, n, a)
          .reversibleMessageMixin();
    msg.setReceiveTime(
      if(isTentative, infinite, void));
    msg.setID(uid());
    msg
  };
```

```
  new(anId, aReference)::copy({
    id:=anId; reference:=aReference });

  notify(content, isTentative)::{
    if(not(isTentative),
       sentbox.asVector().iterate({
         sentbox.delete(el);
         copiedMsg: el.copy();
         copiedMsg.setNegative();
         outbox.add(copiedMsg) }));
    reference#invokeWhen(
      id, content, isTentative) }
});
```

Table 8.16: Implementation of the tentative future observer

```
tentativeFuturesMixin()::{
  invokeWhen(anId, content, isTentative)::{
    whenInfo: whenBlocks.get(anId);
    code       : whenInfo[1];
    reverse    : whenInfo[2];
    prevContent: whenInfo[3];
    isFirstTime: whenInfo[4];
    if(not(is_void(reverse)) &
       not(isFirstTime),
       { reverse(
           prevContent, isTentative) });
    v: code(content, isTentative);
    whenInfo[3]:=content;
    whenInfo[4]:=false;
    v
  };

  createMessage(src, target, name, args)::{
    .createMessage(src, target, name, args)
      .tentativeFutureMessageMixin()
  };
```

```
  when(aFuture,
       code(content, isTentative))::{
    whenReverse(aFuture,
                code(content, isTentative),
                void);
  };

  whenReverse(
    aFuture,
    code(content, isTentative),
    reverse(content, isTentative))::
  { whenBlocks.add(
      [code, reverse, void, true]);
    aFuture#subscribe(
      actor(futureObserver.new(
               newId, thisActor())));
    newId:=newId+1;
    void };

  requires(futuresMixin);
  capture()
};
```

Table 8.17: Implementation of the `tentativeFuturesMixin`

Whereas this is but a first proposal the described protocol illustrates how the first-class mailboxes of the paradigm support the implementation of language constructs that deal with such concurrency conflicts.

Although the first results of experimentation with this language construct are promising, further research is needed to evaluate this approach in a number of applications. It has also not been investigated how reversibility interacts with the other language constructs we proposed in the previous chapter.

8.6 Summary

In this chapter we have further exploited AmbientTalk as a language laboratory and experimented with language constructs addressing two important themes to develop mobile distributed systems. The first theme addressed language constructs that facilitate group communication and coordination in the context of volatile connections. The most important abstraction resulting from this experiment is the multi-future, which represents an unordered set of results yielded by asynchronous group communication. Furthermore, two abstractions were introduced to coordinate the concurrency spawned by these, possibly recursive, asynchronous group communications, to wit the `whenEach` and `whenAll` constructs.

The second theme comprised optimistic concurrency control mechanisms. Optimistic concurrency control strategies are important techniques for programming mobile distributed systems because they best support the autonomous nature of devices and the natural concurrency that results from this autonomy. The most important result was that we were able to enable weak replication at the level of active objects. This result was achieved thanks to the combination of two protocols, namely Time Warp and Bayou's anti-entropy protocol. These two complex protocols extensively rely on the reified communication traces of the AmOP paradigm – both were relatively easily implemented thanks to the extensive use of AmbientTalk's mailboxes. What is more, the language mixin that encapsulates Bayou's anti-entropy protocol was conceived as an extension of the Time Warp mixin. This demonstrates the flexibility of AmbientTalk's mixin methods to compose language mixins together. Finally, we introduced abstractions to deal with tentative data that results from interacting with weakly replicated objects.

Although the abstractions in this chapter are first rough proposals they all rely on the AmOP criteria we distilled in chapter 3 and demonstrate the potential of the AmOP paradigm for programming advanced AmOP applications.

Chapter 9

Conclusion

In the final chapter of this dissertation we take a step back and contemplate our results. We summarize the work presented in this dissertation and situate our work in the context of the three research goals we set out in the introduction. Next, we summarize the contributions made in this dissertation and speculate on areas that are interesting for future research, some of which is already underway.

9.1 Introduction

In this dissertation we have illustrated that programming languages that support the Ambient-Oriented Programming paradigm facilitate programming mobile distributed systems.

The usefulness of the Ambient-Oriented Programming paradigm stems from the fact that it was shaped by the hardware phenomena induced by mobile distributed systems. These hardware phenomena distinguish mobile distributed systems from traditional distributed systems and impose new requirements on the software applications. The Ambient-Oriented Programming paradigm addresses these requirements from the perspective of object-oriented software development.

9.2 Summary and Contributions

Chapters 2 to 8 present the main body of this dissertation and are summarized below.

9.2.1 Restrictions of Existing Software Platforms

In chapter 2 we considered the differences between mobile and traditional distributed systems and from these differences we identified hardware phenomena that distinguish both types of distributed systems. These hardware phenomena were discussed through the context of well-established concurrency and distribution concepts.

We discussed three approaches in order to express concurrency and distribution concepts in the object paradigm. First, in the library approach concurrency and distribution concepts are modeled with objects so that the typical object

composition mechanisms can be used to create the necessary concurrency and distribution abstractions. Second, the integrative approach aligns concepts of the object paradigm with concepts of concurrency and distribution. The advantage of the integrative approach is that the programmer has to deal with fewer concepts so that concurrency and distribution are more naturally dealt with. Third, the reflective approach aims to combine the flexibility of the library approach with the advantages of integration. We found that the integrative approach best matched our goal to unify the requirements of the hardware phenomena with the object paradigm. The integrative approach consists of expressing concurrency and distribution in a distributed object oriented language, whereas the library approach consists of expressing concurrency and distribution in the form of middleware. The reflective approach can be applied in the context of both languages and middleware. Hence, to make the explicit distinction between both the integrative and the library approach we categorized the state of the art to program mobile distributed systems in languages and middleware. None of the distributed languages we discussed provides sufficient support to deal with the hardware phenomena. Nevertheless, we concluded that the languages designed for open networks best preserve the autonomy of devices. Some of the middleware we discussed offered sufficient support to deal with the hardware phenomena. Unfortunately, these approaches did not match the object paradigm very well. For this reason we defined the Ambient-Oriented Programming paradigm which defines programming languages for dealing with the observed hardware phenomena.

9.2.2 Ambient-Oriented Programming

In chapter 3, we took a step back from the problems we encountered in the state of the art. We distilled four characteristics from the hardware phenomena that shape the Ambient-Oriented Programming (AmOP) paradigm. This paradigm starts from a concurrent distributed object-oriented programming language. We argued that the object model for such a language should be classless because a classless object model enables one to control explicitly all sharing relationships between objects. Control over this aspect is necessary because sharing is important from two perspectives. First, from a concurrency perspective it is important in the context of maintaining a consistent state. Second, from a distribution perspective it is important in the context of failures. The other three criteria that define the paradigm shape the concurrency and distribution properties of the object model: first, non-blocking communication preserves the autonomy of devices in the face of concurrency control techniques. Second, reified communication traces are required to ensure that objects can maintain a consistent state in the face of non-blocking communication. Reified communication traces also enable one to devise customized message delivery guarantees. Third, a reified environmental context is needed to make objects aware of their direct ambient so that they can identify ambient resources and sense when ambient resources have become unavailable.

After we distilled these criteria, we revisited the software platforms from chapter 2 in the context of the paradigm to further understand their strengths and weaknesses with respect to programming mobile distributed systems. Distributed languages designed to support open networks adhere to the non-blocking communication primitives criterion and are sometimes based on a classless ob-

ject model. However, none of the languages adhere to the reified communication traces or the reified environmental context criteria. The state of the art in middleware, especially the tuple space based approaches, best support the AmOP criteria but are not based on a classless object model and do not match the object paradigm very well.

9.2.3 An AmOP Concurrency and Distribution Model

The next step in the dissertation is to build a concurrent and distributed language that fits the AmOP paradigm. However, before we can proceed with this step we have to choose a concurrency and distribution model for our language. For our concurrency and distribution model we rely on the actor model because of two reasons. First, the model fits the object paradigm well. Second, the model best matches the AmOP criteria we distilled in chapter 3. Unfortunately, the actor model does not fulfill all four AmOP criteria. For this reason we proposed an extension of the actor model in chapter 4. This extended actor model, which we call the ambient actor model, is defined by means of an extension of the operational semantics of the actor model and introduces the concept of explicit mailboxes. Explicit mailboxes augment the actor model with reified the communication traces and environmental context, which were the two AmOP criteria lacking in the actor model. Afterwards, we wrote a first AmOP application expressed in the model. This limited experiment already demonstrated the usefulness of the first-class mailboxes but also showed us that the language specified by the model is too low-level to express realistic AmOP applications. Therefore, the next step was to build a high-level language based on the ambient actor model.

9.2.4 An AmOP Language: AmbientTalk

In chapter 5, we built a programming language kernel, called AmbientTalk. AmbientTalk is based on the ambient actor model but has a more advanced object system. The object system of the ambient actor model, which was inherited from the standard actor model, does not support state mutations within an object method. To resolve this limitation we were inspired by the ABCL model. Another important design decision was to choose for a double layered object system, consisting of a passive and active object layer. The passive object layer is sequential, whereas the active object layer is concurrent and distributed. This separation of layers has the advantage that it reduces the mental overhead of having to think of all objects as potentially concurrent and distributed ones. The passive object layer was based on Pic%'s object model. Passive objects are similar to traditional sequential objects that communicate synchronously but which are created without classes. Instead, objects are created by cloning them from an existing object or by extending an existing object. Objects can be extended at run-time using predefined mixin methods or using arbitrarily defined extensions outside an object. Another distinctive feature of the passive object layer is its special parameter binding rules. An actual parameter can be bound lazily to a formal parameter. In that case the formal parameter is bound to a first-class method. This type of parameter binding proved to be very useful in the context of extending the language with new language constructs.

The active object layer introduces concurrency and distribution based on the ambient actor model in the language. An active object points to a passive object that defines its behavior. The message passing semantics defines the communication rules between active objects. The most important rules are that communication is always non-blocking and that passive objects that are passed as arguments are deep-copied up to the level of active objects. Deeply copied passive objects prevent concurrency problems that result from sharing passive objects. Hence, only active objects can be shared by other objects. The active object layer inherits first-class messages and mailboxes from the ambient actor model but also adds the notion of mailbox observers. Mailbox observers can monitor changes to the mailboxes. Finally, we further experimented with these concepts in the context of an AmOP application expressed in AmbientTalk.

9.2.5 AmbientTalk as a Language Laboratory

In order to further explore the potential of the AmOP paradigm we have included a meta object protocol (MOP) in AmbientTalk such that it enables experimentation with expressive programming language abstractions. The purpose of these abstractions is to give developers the means to deal explicitly with the hardware phenomena exhibited by mobile distributed systems. We formally defined the MOP in AmbientTalk by means of a metacircular AmbientTalk implementation. The advantages of this approach is that it not only provides a clear definition of the MOP but also of the AmbientTalk language itself. Moreover, it allows us to demonstrate a sense of paradigmatic completeness of both the language and the AmOP paradigm. What is more, the semantics of the MOP are defined in terms of AmbientTalk's first-class mailboxes such that the MOP is aligned with the AmOP paradigm.

We have shown that metaprograms can be used to extend AmbientTalk with new language constructs. These language constructs can be encapsulated in mixin methods, which we called language mixins.

9.2.6 Experiments with Language Constructs

In chapters 7 and 8 we discussed the implementation of several language constructs based on the MOP we specified in chapter 6. We have illustrated for each of these language constructs how they aid in expressing AmOP applications. Furthermore, we have shown that they are expressed using AmbientTalk's MOP and the extent in which they rely on the properties of the AmOP paradigm.

In the second part of chapter 7 we have put some of these language constructs to use in a chat application. Such a chat application epitomizes the concerns that arise when developing an AmOP application. We have compared this AmbientTalk application to a similar application expressed in Java and found that the program expressed in AmbientTalk was a factor of six smaller in terms of lines of code. What is more, the program which was expressed in AmbientTalk better supports the hardware phenomena exhibited by this mobile distributed system.

Finally, in chapter 8 we expressed some more advanced language constructs. More particularly we investigated two topics. First, we discussed group communication and coordination in the context of mobile distributed systems. These

abstractions are important in order to gain insight in the structure of collaborative applications. The second topic was weak replication in the context of the object paradigm. Weak replication is an important abstraction in order to enhance the autonomy of devices but the state of the art only applied it in the context of passive data. We were able to promote weak replication to the active object level by integrating a protocol that enables reversible computations with a weak replica synchronization protocol. Furthermore, we discussed a programming language abstraction that deals with tentative data that arise from weak replication. These abstractions demonstrate the potential of the AmOP paradigm and and its ability to structure advanced AmOP applications.

9.2.7 Conclusion

As was explained in chapter 1, the goal of our research is to a) invent expressive programming language abstractions that facilitate the construction of AmOP applications, b) obtain a better understanding of the structure of future AmOP applications and c) get insight in the fundamental semantic building blocks that lie at the basis of these abstractions in the same vein continuations are the foundations of control flow and environments are at the heart of scoping mechanisms.

Since the conception of AmOP applications is currently still in its infancy, it is hard to do research in a purely demand-driven way. For this reason our research methodology consisted of pursuing these research goals simultaneously such that an interplay between these three goals came into existence: the programming language abstractions determine the set of semantic building blocks. However, the power of the set of semantic building blocks determines the extent of the programming language constructs. These programming language constructs in turn heavily determine the structure of AmOP applications. Finally, when more complex AmOP applications will be developed the need for more expressive programming language constructs will arise.

Bootstrapping this interplay has been based on the unraveling of the hardware phenomena that fundamentally discriminate mobile devices (connected by mobile networks) from classic desktop machines (connected by stationary networks). Based on this unraveling we have defined a number of semantic building blocks in the form of an actor model extension. This extension reifies its actor communication traces and has a continuous explicit causal link with the ambient that reifies an actor's environmental context. These semantic building blocks have been studied at the formal level and have been used in practice to build a number of initial programming language abstractions. Subsequently these programming language abstractions were used to structure an AmOP and this experience gave rise to the development of other programming language abstractions. Nevertheless, because of the interplay between our research goals there is room for further research at all three levels. This is the topic of section 9.3.

9.3 Limitations and Future Work

Now that we have presented what we have achieved, it is time to mention what we did not do. We present a number of directions for future research and opportunities for improvements.

9.3.1 AmbientTalk's Shortcomings

There are a number of rough edges to the AmbientTalk kernel language that need further polishing. The most important ones are described. First, AmbientTalk does not feature a pure object based data model. Data values such as numbers and tables are not represented as objects but rather as primitive values. As a result native functions must be introduced to deal with these values such that they form a special case in the code. A second rough edge that needs work is AmbientTalk's MOP. Language constructs are currently composed of a redefinition of the MOP and public methods that introduce the language construct in an object. Although the language construct can be encapsulated in a mixin method, the messages that are sent in the context of the MOP have the identity of the active object. As a result the interface of an active object becomes "polluted" with methods that handle these messages as more language constructs such that name conflicts between meta and base methods can exist. A possible solution to resolve this problem is the introduction of layers into an active object. Each layer has its own public interface but is still associated with the same active object. When sending a message to an active object one can choose the layer to be addressed. If no specific layer is addressed then the message is directed to the base behavior of an object, otherwise it is handled by the specific layer. Each layer could denote a different meta-behavior of the active object. Nevertheless, research is needed to validate this approach and compare it to other approaches.

9.3.2 Language Constructs

Some of the language constructs presented in this dissertation are tentative and need to be further developed and subjected to further experimentation. For example, in the case of ambient references we noticed that the design dimensions we proposed are not complete. In order to discover other useful dimensions we need to further evaluate them in a wider range of AmOP applications. It also seems that ambient references would benefit from a more expressive discovery mechanism than the one available in AmbientTalk. Group communication and weak replication abstractions also show much potential to reduce the complexity of specific types of AmOP applications. Although both of these abstractions were experimented with in small AmOP applications further experiments are needed to validate the expressiveness and their scalability. This is currently being investigated by a member of our lab.

Another interesting field of future work is to see how transaction management in classic distributed systems can be transposed to the AmOP setting in which devices holding a lock may leave and never return. Reified communication traces may once again prove useful here, as already exemplified by optimistic process collaboration approaches such as the Time Warp mechanism [Jef85] we discussed in section 8.3.

In the model presented so far, no attention was given to exception handling features. Exception handling mechanisms define a context in which dynamically raised exceptions are to be handled. In an ambient-oriented setting, such contexts cannot be described using classic try-catch blocks since the processes making up an ambient-oriented application communicate using non-blocking communication primitives. This implies that the calling process may have left

the context of its try block before the exception was propagated by the invoked process. The exception handling mechanisms in E [MTS05] might be a good starting point. This is currently being investigated by a member of our lab.

9.3.3 Integration of Language Constructs

As Hoare noted in his paper "Hints on Programming Language Design" [Hoa73] it is important not to fall into the trap of feature piling when designing a programming language. Rather, before adding a new language feature to a language one has to reflect so as to see whether it is possible to realize the semantics of that feature by unifying existing concepts. In the AmOP language constructs we have attempted to do this on several occasions. Namely, in several experiments we have considered the concept of futures to structure AmOP applications. We have first introduced futures in AmbientTalk for aligning non-blocking communication with data-flow computations. We considered futures again as an abstraction for the coordination of group communication. In this context futures were thought of as a set of results rather than a single value. In the context of weak replication we have thought of futures as a possibly tentative result that is transparently resolved again with new results until the result becomes stable and is considered to be definitive. Ultimately, ambient references could be regarded as the futures of service discovery. In this perspective an ambient reference represents an active object that is to be discovered in the future. It is clear that the concept of futures plays a prominent role for structuring AmOP applications. From an integration perspective it is however not clear how we can unify these different types of futures into a single concept that has clean and simple semantics and does not introduce other conflicts. Perhaps a taxonomy that describes futures from a different perspective can provide further insight into finding a "unified" future abstraction.

The interactions between language features from a technical perspective also has to be further investigated. The different forms of scoped reflection and the use of language mixins to encapsulate language constructs offers some initial support to control the interactions between language features. However, all in all it is a difficult exercise to predict how two arbitrary language mixins will behave when they are composed. The interactions between language constructs have to be considered per case. Hence, more advanced meta-level techniques are necessary to unravel the combined semantics of language constructs and resolve possible conflicts.

9.3.4 Efficient Implementation

A non-recursive prototype implementation of AmbientTalk has been built that supports experimentation with AmOP language constructs. However, during the conception of AmbientTalk not much attention was paid to efficient execution both in time and in space. More insight is required on how to map AmOP features onto an efficient implementation technology. For instance, new distributed memory management techniques are required because existing techniques are not intended for use in mobile networks. We predict that automated distributed memory management systems are unfeasible in this setting because there is no single strategy to deal with volatile connections. We foresee that semi-automatic memory management systems are a possible strategy because

they allow one to specify application-specific rules to reclaim memory, while preventing having to think of memory issues for every single object. This is currently being investigated by a member of our lab. Another example is object serialization for classless object systems. An advantage of serialization in the context of a class-based object system is that the identity of a class can be used to reduce network traffic. In this case the object's class is transparently rebound such that the class does not have to be sent along with the object. In the case of a classless object system this is impossible because there is no notion of a class. As was discussed in section 5.3.6, at the memory level classless objects also share behavior between cloned objects but without making these sharing relationships visible at the language level. These techniques should be transposed to the context of serialization and rebinding. Perhaps the implementation techniques that introduce the concept of cloning families in Kevo [Tai93] might be a starting point.

9.3.5 Security

Security is an important concern in the context of open networks that has not been considered in the context of this dissertation. There are many aspects to ensure security going from low-level encrypted protocols to high-level concerns addressed at the language level. At the language level we have discussed how AmbientTalk's objects can be protected in the face of inheritance such that their encapsulation is preserved. Preserving the encapsulation is an important requirement for all capability-based security models. These capability-based models have been employed successfully in the context of open distributed systems [Mil04]. Nevertheless, it is necessary to investigate how these capability-based models scale in the context of mobile distributed systems.

Appendix A

Code Listing of Metacircular AmbientTalk

A.1 Scanner

```
{

  AOP_token::  1;
  CAT_token::  2;
  CCL_token::  3;
  CEQ_token::  4;
  COL_token::  5;
  COM_token::  6;
  END_token::  7;
  FRC_token::  8;
  LBC_token::  9;
  LBR_token:: 10;
  LPR_token:: 11;
  MOP_token:: 12;
  NAM_token:: 13;
  NBR_token:: 14;
  PER_token:: 15;
  QUO_token:: 16;
  RBC_token:: 17;
  RBR_token:: 18;
  ROP_token:: 19;
  RPR_token:: 20;
  SMC_token:: 21;
  TXT_token:: 22;
  XOP_token:: 23;
  SHA_token:: 23;

Scanner() :: {

  _MSG_:: [ "additive operator",
            "application",
            "declaration",
            "assignment",
            "definition",
            "comma",
            "end of text",
            "fraction",
            "left brace",
            "left bracket",
            "left parenthesis",
            "multiplicative operator",
            "name",
            "number",
            "period",
            "quotation" ,
            "right brace",
            "right bracket",
            "relational operator",
            "right parenthesis",
            "semicolon",
            "text",
            "exponentiation operator",
            "sharp"];

  _SCAN_: void;

      aop::  1;
      apo::  2;
      bkq::  3;
      cat::  4;
```

```
col::  5;
com::  6;
dgt::  7;
eol::  8;
eql::  9;
exp:: 10;
ill:: 11;
lbc:: 12;
lbr:: 13;
lpr:: 14;
ltr:: 15;
mns:: 16;
mop:: 17;
nul:: 18;
per:: 19;
pls:: 20;
quo:: 21;
rbc:: 22;
rbr:: 23;
rop:: 24;
rpr:: 25;
smc:: 26;
wsp:: 27;
xop:: 28;
sha:: 29;

NBR:: 29;
NUL::  0;

ch_tab:: [`nul` ill, ill, ill, ill, ill, ill, ill,
           ill, wsp, eol, ill, ill, eol, ill, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
           wsp, xop, quo, sha, aop, aop, mop, apo,
           lpr, rpr, mop, pls, com, mns, per, mop,
           dgt, dgt, dgt, dgt, dgt, dgt, dgt, dgt,
           dgt, dgt, col, smc, rop, eql, rop, xop,
           cat, ltr, ltr, ltr, ltr, exp, ltr, ltr,
           ltr, ltr, ltr, ltr, ltr, ltr, ltr, ltr,
           ltr, ltr, ltr, ltr, ltr, ltr, ltr, ltr,
           ltr, ltr, ltr, lbr, mop, rbr, xop, ltr,
           bkq, ltr, ltr, ltr, ltr, exp, ltr, ltr,
           ltr, ltr, ltr, ltr, ltr, ltr, ltr, ltr,
           ltr, ltr, ltr, ltr, ltr, ltr, ltr, ltr,
           ltr, ltr, ltr, lbc, aop, rbc, rop, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
           ill, ill, ill, ill, ill, ltr, ill, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
           ill, ill, ill, ill, ill, ill, ill, ill,
```

```
          ill, ill, ill, ill, ill, ill, ill, ill ];

INP: void;
SIZ: void;
POS: void;
HLD: void;
CHR: void;

skip_ch()::
  CHR:= if((POS:= POS+1) > SIZ,
           NUL,
           INP[POS]);

get_cat()::
  if(CHR = NUL, nul, ch_tab[ord(CHR)]);

next_ch(Tkn)::
  { skip_ch();
    Tkn };

freeze()::
  HLD:= POS-1;

capture_name(Tkn)::
  { t[POS-HLD-1]: INP[HLD:= HLD+1];
    _SCAN_:= implode(t);
    Tkn };

capture_text(Tkn)::
  { t: if(POS-HLD > 2,
          v[POS-HLD-2]: INP[HLD:= HLD+1],
          []);
    _SCAN_:= implode(t);
    Tkn };

capture_number(Tkn)::
  { t[POS-HLD-1]: INP[HLD:= HLD+1];
    _SCAN_:= number(implode(t));
    Tkn };

check(allowed)::
  allowed[get_cat()];

mask@list::
  { msk[NBR]: false;
    for(k: 1, k <= size(list), k:= k+1,
      msk[list[k]]:= true);
    msk };

bkq_allowed:: mask(bkq);
col_allowed:: mask(col);
dgt_allowed:: mask(dgt);
eql_allowed:: mask(eql);
exp_allowed:: mask(exp);
nam_allowed:: mask(dgt,exp,ltr);
opr_allowed:: mask(aop,eql,mns,mop,pls,rop,xop);
per_allowed:: mask(per);
quo_allowed:: mask(quo);
qux_allowed:: mask(eol,nul,quo);
sgn_allowed:: mask(mns,pls);
```

```
operator(Tkn)::
  { freeze();
    until(!check(opr_allowed), skip_ch());
    capture_name(Tkn) };

exponent()::
  { skip_ch();
    if(check(sgn_allowed), skip_ch());
    if(!check(dgt_allowed), Error("digit required"));
    until(check(!dgt_allowed), skip_ch());
    capture_number(FRC_token) };

fraction()::
  { skip_ch();
    if(!check(dgt_allowed), Error("digit required"));
    until(!check(dgt_allowed), skip_ch());
    if(check(exp_allowed),
      exponent(),
      capture_number(FRC_token)) };

text()::
  { while(!check(qux_allowed), skip_ch());
    if(check(quo_allowed),
      skip_ch(),
      Error("quote required")) };

aop_fun()::
  operator(AOP_token);

apo_fun()::
  next_ch(QUO_token);

bkq_fun()::
  { skip_ch();
    while(!check(bkq_allowed), skip_ch());
    skip_ch();
    Scan() };

cat_fun()::
  next_ch(CAT_token);

col_fun()::
  { skip_ch();
    if(check(eql_allowed),
      next_ch(CEQ_token),
      if(check(col_allowed),
        next_ch(CCL_token),
        COL_token)) };

com_fun()::
  next_ch(COM_token);

dgt_fun()::
  { freeze();
    until(!check(dgt_allowed), skip_ch());
    if(check(per_allowed),
      fraction(),
      if(check(exp_allowed),
        exponent(),
```

```
        capture_number(NBR_token))) };

ill_fun()::
  { Error("illegal character");
    END_token };

lbc_fun()::
  next_ch(LBC_token);

lbr_fun()::
  next_ch(LBR_token);

lpr_fun()::
  next_ch(LPR_token);

ltr_fun()::
  { freeze();
    until(!check(nam_allowed), skip_ch());
    capture_name(NAM_token) };

mop_fun()::
  operator(MOP_token);

nul_fun()::
  next_ch(END_token);

per_fun()::
  next_ch(PER_token);

quo_fun()::
  { skip_ch();
    freeze();
    text();
    capture_text(TXT_token) };

rbc_fun()::
  next_ch(RBC_token);

rbr_fun()::
  next_ch(RBR_token);

rop_fun()::
  operator(ROP_token);

rpr_fun()::
  next_ch(RPR_token);

smc_fun()::
  next_ch(SMC_token);

wsp_fun()::
  { skip_ch();
    Scan() };

xop_fun()::
  operator(XOP_token);

sha_fun()::
  next_ch(SHA_token);
```

```
    fun_tab:: [ aop_fun,
                apo_fun,
                bkq_fun,
                cat_fun,
                col_fun,
                com_fun,
                dgt_fun,
                wsp_fun,
                rop_fun,
                ltr_fun,
                ill_fun,
                lbc_fun,
                lbr_fun,
                lpr_fun,
                ltr_fun,
                aop_fun,
                mop_fun,
                nul_fun,
                per_fun,
                aop_fun,
                quo_fun,
                rbc_fun,
                rbr_fun,
                rop_fun,
                rpr_fun,
                smc_fun,
                wsp_fun,
                xop_fun,
                sha_fun ];

    Scan() :: { _SCAN_:= void;
                fun_tab[get_cat()]() };

    initScan(Str) ::
      { INP := explode(Str);
        SIZ := size(INP);
        POS := 0;
        skip_ch();
        void };

    getAttribute() :: _SCAN_;

    capture() };

 _SCANNER_ :: Scanner();

display("scanner ...... installed", eoln) }
```

A.2 Parser

```
{
   REF(txt) :: agReferenceP.cloneMe(txt);
   TXT(str) :: agTextP.cloneMe(str);
   FUN(nam,args,bdy) :: agFunctionP.cloneMe(nam,args,bdy);
   TAB(tb) :: agTableP.cloneMe(tb);
   NBR(n) :: agNumberP.cloneMe(n);
   FRC(f) :: agFractionP.cloneMe(f);
   VOI() :: agVoidP;
   BND(nam,val,nxt) :: agBindingP.cloneMe(nam,val,nxt);
   OBJ(cst,var,nxt) :: agObjectP.cloneMe(cst,var,nxt);
   APL(nam,arg) :: agApplicationP.cloneMe(nam,arg);
   TBL(nam,idx) :: agTabulationP.cloneMe(nam,idx);
   MSG(exp,inv) :: agSyncMessageP.cloneMe(exp,inv);
   ASY(exp,inv) :: agAsyncMessageP.cloneMe(exp,inv);
   SUP(inv) :: agSuperP.cloneMe(inv);
   DEF(inv,exp) :: agDefinitionP.cloneMe(inv, exp);
   DCL(inv,exp) :: agDeclarationP.cloneMe(inv, exp);
   SET(inv,exp) :: agAssignmentP.cloneMe(inv, exp);
   QUO(exp) :: agQuoteP.cloneMe(inv, exp);
   CLO(fun,env) :: agCloneP.cloneMe(fun, env);
   CTX(cur,ths,sup) :: agValueP.agContext(cur,ths,sup);
   NAT(idx) :: agNativeP.cloneMe(idx);

   Parser():
    { begin_ref:: REF(TXT("begin"));
      table_ref:: REF(TXT("table"));

      TKN: void;

      skip()::
        TKN:= _SCANNER_.Scan();

      next(Itm)::
        { skip(); Itm };

      unexpected@any::
        Error("Unexpected " + _SCANNER_._MSG_[TKN]);

      expect(Tkn)::
        if(TKN = Tkn, skip(), unexpected());

      infix(Opr, Tkn)::
        { loop(opr, count):
            { opd: Opr();
              if((TKN = Tkn) & (opr = _SCANNER_.getAttribute()),
                 tab: { skip();
                        loop(opr, count+1) },
                 tab[count]: void);
              tab[count]:= opd;
              tab };
          opd: Opr();
          while(TKN = Tkn,
            { opr: next(_SCANNER_.getAttribute());
              arg: loop(opr, 2);
              arg[1]:= opd;
              opd:= APL(REF(TXT(opr)), TAB(arg)) });
          opd };

      list(Sep, Trm)::
        { loop(count):
```

```
      { exp: expression();
        if(TKN = Sep,
           tab: { skip();
                  loop(count+1) },
           if(TKN = Trm,
              next(tab[count]: void),
              unexpected()));
        tab[count]:= exp;
        tab };
    TAB(loop(1)) };

identity(Exp)::
  Exp;

number()::
  NBR(next(_SCANNER_.getAttribute()));

fraction()::
  FRC(next(_SCANNER_.getAttribute()));

text()::
  TXT(next(_SCANNER_.getAttribute()));

quotation()::
  { skip();
    exp: expression();
    QUO(exp) };

variable()::
  REF(TXT(next(_SCANNER_.getAttribute())));

var_case:: case(NAM_token ++ variable,
                AOP_token ++ variable,
                MOP_token ++ variable,
                ROP_token ++ variable,
                XOP_token ++ variable,
                     void ++ unexpected);

application(Exp)::
  APL(next(Exp), if(TKN = RPR_token,
                    next(_EMPTY_),
                    list(COM_token, RPR_token)));

apply(Exp)::
  APL(next(Exp), operand());

tabulation(Exp)::
  TBL(next(Exp), list(COM_token, RBR_token));

qua_case:: case(LPR_token ++ application,
                CAT_token ++ apply,
                LBR_token ++ tabulation,
                     void ++ identity);

qualification(Exp)::
  { skip();
    cas: var_case(TKN);
    ref: cas();
    cas:= qua_case(TKN);
    MSG(Exp, cas(ref)) };
```

```
async(Exp)::
  { skip();
    cas: var_case(TKN);
    ref: cas();
    cas:= qua_case(TKN);
    ASY(Exp, cas(ref)) };

void_(Exp)::
  void;

inv_case:: case(LPR_token ++ application  ,
                CAT_token ++ apply        ,
                LBR_token ++ tabulation   ,
                PER_token ++ qualification,
                SHA_token ++ async        ,
                     void ++ void_        );

invocation(Exp)::
  { cas: inv_case(TKN);
    exp: cas(Exp);
    if(is_void(exp),
      Exp,
      invocation(exp)) };

name()::
  invocation(variable());

begin_()::
  { skip();
    arg: list(SMC_token, RBC_token);
    exp: APL(begin_ref, arg);
    invocation(exp) };

table()::
  { skip();
    arg: if(TKN = RBR_token,
            next(_EMPTY_),
            list(COM_token, RBR_token));
    exp: APL(table_ref, arg);
    invocation(exp) };

subexpression()::
  { skip();
    exp: expression();
    expect(RPR_token);
    invocation(exp) };

operator: void;

unr_case:: case(NBR_token ++ number   ,
                FRC_token ++ fraction ,
                TXT_token ++ text     ,
                QUO_token ++ quotation,
                AOP_token ++ operator ,
                MOP_token ++ operator ,
                ROP_token ++ operator ,
                XOP_token ++ operator ,
                NAM_token ++ name     ,
                LBC_token ++ begin_   );
```

```
unary(Ref)::
  { cas: unr_case(TKN);
    exp: cas();
    APL(Ref, TAB([exp])) };

opr_case:: case(NBR_token ++ unary     ,
                FRC_token ++ unary     ,
                TXT_token ++ unary     ,
                QUO_token ++ unary     ,
                AOP_token ++ unary     ,
                MOP_token ++ unary     ,
                ROP_token ++ unary     ,
                XOP_token ++ unary     ,
                NAM_token ++ unary     ,
                LBC_token ++ unary     ,
                     void ++ invocation);

operator():=
  { ref: variable();
    cas: opr_case(TKN);
    cas(ref) };

opd_case:: case(NBR_token ++ number       ,
                FRC_token ++ fraction     ,
                TXT_token ++ text         ,
                QUO_token ++ quotation    ,
                AOP_token ++ operator     ,
                MOP_token ++ operator     ,
                ROP_token ++ operator     ,
                XOP_token ++ operator     ,
                NAM_token ++ name         ,
                LBC_token ++ begin_       ,
                LBR_token ++ table        ,
                LPR_token ++ subexpression,
                     void ++ unexpected   );

operand()::
  { cas: opd_case(TKN);
    cas() };

factor()::
  infix(operand, XOP_token);

term()::
  infix(factor, MOP_token);

comparand()::
  infix(term, AOP_token);

operation()::
  infix(comparand, ROP_token);

definition(Opr)::
  DEF(next(Opr), expression());

declaration(Opr)::
  DCL(next(Opr), expression());

assignment(Opr)::
  SET(next(Opr), expression());
```

```
    exp_case:: case(COL_token ++ definition ,
                    CCL_token ++ declaration,
                    CEQ_token ++ assignment ,
                         void ++ identity   );

    expression()::
      { opr: operation();
        cas: exp_case(TKN);
        cas(opr) };

    Read(Str)::
      { _SCANNER_.initScan(Str);
        TKN := _SCANNER_.Scan();
    exp: expression();
        expect(END_token);
        exp };

    capture() };

 _PARSER_ :: Parser();

display("reader ....... installed", eoln) }
```

A.3 Abstract Grammar

```
{

Error@Msg::
{ display("***error*** ");
  display@Msg;
  continue(_EXIT_, void) };

Map(Fun, Tab)::
{ n:: size(Tab);
  k: 0;
  tab[n]:: Fun(Tab[k:=k+1]);
  tab };

AbstractGrammar() :: {

  eval(e) :: void;
  apply(args, e) :: Error("cannot apply a ", this().print(e));
  define(exp, e) :: Error("cannot define a ", this().print(e));
  declare(exp, e) :: Error("cannot declare a ", this().print(e));
  assign(exp, e) :: Error("cannot assign a ", this().print(e));
  syncmessage(dct, e) ::
    Error("illegal sync message expression: ", this().print(e));
  asyncmessage(anActor, e) ::
    Error("illegal async message expression: ", this().print(e));
  supersend(e) :: Error("illegal supersend: ", this().print(e));
  wrap(e) :: this();
  print(e) :: "AbstractGrammar";
  getMetaValue() :: this();

  call(dct,acts,e) ::
    Error("call: illegal function parameter: ", this().print(e));
  bind(dct,act,e)  ::
    Error("bind: illegal formal parameter: ", this().print(e));

  isVoid()     :: false;
  isFraction() :: false;
  isText()     :: false;
  isTable()    :: false;
  isNative()   :: false;
  isBinding()  :: false;
  isObject()   :: false;
  isCallFrame():: false;
  isFunction() :: false;
  isNumber()   :: false;
  isClosure()  :: false;
  isActor()    :: false;
  isMailbox()  :: false;

  isNumeral() :: this().isNumber()||this().isFraction();
  isApplicable() :: this().isClosure();
  isDictionary() :: this().isCallFrame()||this().isObject();

  printBrackets(val) :: "<"+val+">";

  agReference(name) :: {
    getName()    :: name;
    setName(nam) :: name := nam;
    cloneMe(nam) :: { c: this().clone(root); c.setName(nam); c};

    eval(e)      :: { v: e.cur.lookupAny(name, e.ths, e); v };
```

```
  define(exp, e) :: { e.cur.addVariable(name, v:exp.eval(e) ); v };
  declare(exp,e) :: { e.cur.addConstant(name, v:exp.eval(e) ); v };
  assign(exp, e) :: { e.cur.setVariable(name, v:exp.eval(e) ); v };
  syncmessage(dct, e) :: { dct.lookupConstant(name, dct, e) };
  asyncmessage(anActor, e) :: {
    `return message(e.thsActor.getAddress(), anActor, name)`
    actorBehavior: e.cur.agActorBehavior(e);
    actorBehavior.createMessage(
      e.thsActor.getAddress(),
      anActor,
      name,
      agTableP) };
  supersend(e) :: e.sup.lookupAny(name, e.ths, e);

  call(dct,actuals,e) :: {
    actT: actuals.getTable();
    i:0;
    evaluatedActs[size(actT)]: actT[i:=i+1].eval(e);
    dct.addVariable(name, agTableP.cloneMe(evaluatedActs))
  };

  bind(dct,act,e) :: { dct.addVariable(name,act.eval(e)) };

  print(e) :: printBrackets("reference "+name.getTxt());
  capture() `<- agReference`
};

agApplication(expr, args) :: {
  getName()    :: expr;
  getArgs()    :: args;
  setName(nam) :: expr := nam;
  setArgs(arg) :: args := arg;
  cloneMe(nam,arg) :: {
     c: this().clone(root);
     c.setName(nam); c.setArgs(arg);
     c
  };

  eval(e)        :: expr.eval(e).apply(args, e);
  define(exp, e) :: {
    e.cur.addVariable(
      expr.getName(),
      v:agFunctionP.cloneMe(expr, args, exp));
    v.wrap(e) };
  declare(exp,e) :: {
    e.cur.addConstant(
      expr.getName(),
      v:agFunctionP.cloneMe(expr, args, exp));
      v.wrap(e) };
  assign(exp, e) :: {
    e.cur.setVariable(
      expr.getName(),
      v:agFunctionP.cloneMe(expr, args, exp));
    v.wrap(e) };
  syncmessage(dct, e)::
    dct.lookupConstant(expr.getName(), dct, e).apply(args,e);
  asyncmessage(anActor, e) :: {
    `invoke send(aMsg)`
    actorBehavior: e.cur.agActorBehavior(e);
    aMsg: actorBehavior.createMessage(
```

```
      e.thsActor.getAddress(),
      anActor,
      expr.getName(),
      args);
    actorBehavior.send(aMsg) };
  supersend(e)   ::
    e.sup.lookupAny(expr.getName(), e.ths, e).apply(args, e);

  call(dct,actuals,e) :: {
    actT: actuals.getTable();
    i:0;
    closures[size(actT)]:
      agClosureP.cloneMe(agFunctionP.cloneMe(expr,args,actT[i:=i+1]),e);
    dct.addVariable(expr.getName(), agTableP.cloneMe(closures))
  };

  bind(dct, act, e) ::
    dct.addVariable(
      expr.getName(),
      agClosureP.cloneMe(agFunctionP.cloneMe(expr,args,act),e));

  print(e) :: printBrackets("application");
  capture() `<- agApplication`
};

agTabulation(expr, idx) :: {
  getName()    :: expr;
  getIdx()     :: idx;
  setName(nam) :: expr := nam;
  setIdx(id)   :: idx := id;
  cloneMe(nam,id) :: {
     c: this().clone(root);
     c.setName(nam); c.setIdx(id);
     c
  };

  eval(e)      :: expr.eval(e).get(idx.getTbl()[1].eval(e).getMetaValue());
  define(exp, e) :: {
    tab[idx.getTbl()[1].eval(e).getMetaValue()] : exp.eval(e);
    e.cur.addVariable(expr.getName(), v:agTableP.cloneMe(tab) ); v };
  declare(exp, e) :: {
    tab[idx.getTbl()[1].eval(e).getMetaValue()] : exp.eval(e);
    e.cur.addConstant(expr.getName(), v:agTableP.cloneMe(tab) ); v };
  assign(exp, e) :: {
    tab: expr.eval(e);
    tab.set(idx.getTbl()[1].eval(e).getMetaValue(),v:exp.eval(e)); v };
  syncmessage(dct, e) ::
    dct.lookupConstant(expr.getName(), dct, e).get(
      idx.getTbl()[1].eval(e).getMetaValue());
  supersend(e) ::
    e.sup.lookupAny(expr.getName(), e.ths, e).get(
      idx.getTbl()[1].eval(e).getMetaValue());
  print(e) :: printBrackets("tabulation");
  capture() `<- agTabulation`
};

agDefinition(inv, exp) :: {
  getExp() :: exp;
  getInv() :: inv;
  setExp(ex) :: exp := ex;
```

```
  setInv(in) :: inv := in;
  cloneMe(in,ex) :: {
    c: this().clone(root);
    c.setExp(ex); c.setInv(in);
    c };

  eval(e) :: inv.define(exp,e);
  print(e) :: printBrackets("definition");
  capture() `<- agDefinition`
};

agDeclaration(inv, exp) :: {
  getExp() :: exp;
  getInv() :: inv;
  setExp(ex) :: exp := ex;
  setInv(in) :: inv := in;
  cloneMe(in,ex) :: {
    c: this().clone(root);
    c.setExp(ex); c.setInv(in);
    c };

  eval(e) :: inv.declare(exp,e);
  print(e) :: printBrackets("declaration");
  capture() `<- agDeclaration`
};

agAssignment(inv, exp) :: {
  getExp() :: exp;
  getInv() :: inv;
  setExp(ex) :: exp := ex;
  setInv(in) :: inv := in;
  cloneMe(in,ex) :: {
    c: this().clone(root);
    c.setExp(ex); c.setInv(in);
    c };

  eval(e) :: inv.assign(exp,e);
  print(e) :: printBrackets("assignment");
  capture() `<- agAssignment`
};

agSyncMessage(exp, inv) :: {
  getExp() :: exp;
  getInv() :: inv;
  setExp(ex) :: exp := ex;
  setInv(in) :: inv := in;
  cloneMe(ex,in) :: {
    c: this().clone(root);
    c.setExp(ex); c.setInv(in);
    c };

  eval(e) :: inv.syncmessage(exp.eval(e),e);
  print(e) :: printBrackets("sync message");
  capture() `<- agSyncMessage`
};

agAsyncMessage(exp, inv) :: {
  getExp() :: exp;
  getInv() :: inv;
  setExp(ex) :: exp := ex;
```

```
  setInv(in) :: inv := in;
  cloneMe(ex,in) :: {
    c: this().clone(root);
    c.setExp(ex); c.setInv(in);
    c };

  eval(e) :: inv.asyncmessage(exp.eval(e),e);
  print(e) :: printBrackets("async message");
  capture() `<- agAsyncMessage`
};

agSuper(inv) :: {
  getInv() :: inv;
  setInv(in) :: inv := in;
  cloneMe(in) :: {
    c: this().clone(root);
    c.setInv(in);
    c };

  eval(e) :: inv.supersend(e);
  print(e) :: printBrackets("super send");
  capture() `<- agSuper`
};


agValue() :: {
  eval(e) :: this();
  print(e) :: "Value";

  agVoid() :: {
    print(e) :: "void";
    clone(x) :: this();

    picoClone@args :: this();

    lookupConstant(nam, ths, e) ::
      Error("Constant not found: "+nam.getTxt());
    containsConstant(nam, ths, e) :: false;
    lookupAny(nam, ths, e) ::
      Error("Name not found: "+nam.getTxt());
    setVariable(nam, ths) ::
      Error("Variable not found: "+nam.getTxt());
    addVariable(nam, ths) ::
      Error("Cannot add variable: "+nam.getTxt());
    addConstant(nam, ths) ::
      Error("Cannot add constant: "+nam.getTxt());

    debug() :: void;

    lookup(nam,ths,myDct,e) :: void;
    getMetaValue() :: void;
    change(n,v) :: void;
    isVoid() :: true;

    capture() `<- agVoid`
  };

  agBinding(nam,val,nxt) :: {
    getVal() :: val; setVal(v) :: val := v;
    getNam() :: nam; setNam(n) :: nam := n;
```

```
  getNxt() :: nxt; setNxt(n) :: nxt := n;
  cloneMe(n,v,ne) :: {
     c: this().clone(root);
     c.setVal(v); c.setNam(n); c.setNxt(ne);
     c };
  print(e) :: printBrackets("binding");

  picoClone() :: cloneMe(nam,val,nxt.picoClone());
  lookup(n, ths, myDct, e) :: {
    if(n.getTxt()=nam.getTxt(),
       val.wrap(agValueP.agContext(myDct,
                                   ths,
                                   myDct.parent(),
                                   e.thsActor)),
       nxt.lookup(n, ths, myDct, e)) };
  change(n, v) :: if(n.getTxt()=nam.getTxt(),
                     val := v,
                     nxt.change(n,v));
  isBinding() :: true;
  capture() `<- agBinding`
};

agObject(cst,var,nxt) :: {
  getCst() :: cst; setCst(c) :: cst := c;
  getVar() :: var; setVar(v) :: var := v;
  getNxt() :: nxt; setNxt(n) :: nxt := n;
  cloneMe(cs,v,n) :: {
     c: this().clone(root);
     c.setCst(cs); c.setVar(v); c.setNxt(n);
     c };

  parent() :: nxt;
  picoClone(upTo) ::
     this().cloneMe(cst,
                    var.picoClone(),
                    if(this()~upTo,
                       nxt,
                       nxt.picoClone(upTo)));

  containsConstant(nam, ths, e) ::
    if((v:cst.lookup(nam, ths, this(), e))!~void,
       true,
       nxt.containsConstant(nam, ths, e));

  lookupConstant(nam, ths, e) ::
    if((v:cst.lookup(nam, ths, this(), e))!~void,
       v,
       nxt.lookupConstant(nam, ths, e));

  lookupAny(nam, ths, e)   :: {
    found: if ((v:cst.lookup(nam, ths,this(), e))~void,
               var.lookup(nam,ths,this(), e),
               v);
    if(found~void,
       if(nxt~void,
          Error("Could not find variable: ", nam.getTxt()),
          nxt.lookupAny(nam, ths, e)),
       found) };

  setVariable(nam, v) ::
```

```
     if(var.change(nam,v)~void,
        if(nxt~void,
           Error("Could not set variable: "+nam),
           nxt.setVariable(nam,v)),
        v);

addVariable(nam, v) :: {
  var := agBindingP.cloneMe(nam, v, var); this() };
addConstant(nam, v) :: {
  cst := agBindingP.cloneMe(nam, v, cst); this() };

addFrame() :: this().cloneMe(agVoidP, agVoidP, this());
isObject() :: true;

print(e) :: printBrackets("object");
debug() :: {
  `display("constants", eoln);`
  `cst.debug();`
  display("variables", eoln);
  var.debug();
  display("NEXT", eoln);
  nxt.debug()
};

agActorMessage(context)::{
  sourceMethodName:: agTextP.cloneMe("getSource");
  targetMethodName:: agTextP.cloneMe("getTarget");
  nameMethodName  :: agTextP.cloneMe("getName");
  argsMethodName  :: agTextP.cloneMe("getArgs");
  setArgsMethodName  :: agTextP.cloneMe("setArgs");

  setContext(aContext) :: { context:=aContext };
  getContext() :: { context };

  getSource() :: {
    closure: this().lookupConstant(
               sourceMethodName, super(), context);
    closure.apply(agTableP, context)
  };

  getTarget() :: {
    closure: this().lookupConstant(
               targetMethodName, super(), context);
    closure.apply(agTableP, context)
  };

  getName() :: {
    closure: this().lookupConstant(
               nameMethodName, super(), context);
    closure.apply(agTableP, context)
  };

  getArgs() :: {
    closure: this().lookupConstant(
               argsMethodName, super(), context);
    closure.apply(agTableP, context)
  };

  setArgs(anArgsList) :: {
    closure: this().lookupConstant(
```

```
                setArgsMethodName, super(), context);
      closure.apply(agTableP.cloneMe([ anArgsList ]), context)
    };

    capture() `<- agActorMessage`

  };

  agActorBehavior(context)::{
    sendMethodName:: agTextP.cloneMe("send");
    processMethodName:: agTextP.cloneMe("process");
    createMethodName:: agTextP.cloneMe("createMessage");

    setContext(aContext) :: { context:=aContext };
    getContext() :: { context };

    createMessage(src, target, name, args) :: {
      closure: context.cur.lookupConstant(
        createMethodName,
        context.cur,
        context);
      closure.apply(agTableP.cloneMe([src, target, name, args ]),
                    context)
    };

    send(aMsg) :: {
      closure: this().lookupConstant(
                sendMethodName, super(), context);
      closure.apply(agTableP.cloneMe([aMsg]), context)
    };

    process(aMsg) :: {
      closure: this().lookupConstant(processMethodName, super(), context);
      closure.apply(agTableP.cloneMe([aMsg]), context)
    };

    capture() `<- agActorBehavior`

  };

  capture() `<- agObject`
};

agActor(act) :: {
  getAct() :: act; setAct(anAct) :: act := anAct;
  cloneMe(anActorBehavior) :: {
     act: actor(metaActorBehavior.new(anActorBehavior));
     c: this().clone(root);
     c.setAct(act);
     act#initialize(c);
     c };
  isActor() :: true;
  getMetaValue() :: act;
  print(e) :: printBrackets(text(act));
  capture() `<- agActor`
};

agMailbox(contents, addObservers, deleteObservers) :: {
  getContents() :: contents;
  setContents(aVector) :: contents:=aVector;
```

```
  setAddObservers(aVector) :: addObservers := aVector;
  setDeleteObservers(aVector) :: deleteObservers := aVector;
  id: "prototype"; setId(a)::id:=a;
  cloneMe(aName) :: {
     c: this().clone(root);
     c.setContents(vector.new());
     c.setAddObservers(vector.new());
     c.setDeleteObservers(vector.new());
     c.setId(aName);
     c };

  length() :: contents.length();
  add(element) :: {
    contents.add(element);
    addObservers.iterate({ el(element) }) };
  delete(nr) :: {
    element: contents.delete(nr);
    deleteObservers.iterate({ el(element) }) };
  get(nr) :: contents.get(nr);
  set(nr, value) :: contents.set(nr, value);
  iterate(it(el)) :: contents.iterate(it(el));
  select(pred(el)) :: contents.select(pred(el));
  findFirst(pred(el)) :: contents.findFirst(pred(el));
  detect(pred(el)) :: contents.detect(pred(el));
  contains(aValue) :: contents.contains(aValue);
  map(mp(el)) :: contents.map(mp(el));
  remove(element) :: {
    v: contents.remove(element);
    if(v, deleteObservers.iterate({ el(element) }));
    v
  };
  addSyncAddObserver(notify(el)) :: { addObservers.add(notify) };
  addSyncDeleteObserver(notify(el)) :: deleteObservers.add(notify);

  isMailbox() :: true;
  getMetaValue() :: contents;
  print(e) :: printBrackets("mailbox");
  capture() `<- agMailbox`
};

agNative(nat, nam) :: {
  getNat() :: nat; setNat(n) :: nat := n;
  getNam() :: nam; setNam(n) :: nam := n;
  cloneMe(nt, nm) :: {
     c: this().clone(root);
     c.setNat(nt); c.setNam(nm);
     c };
  apply(args, e) :: nat(args, e);
          wrap(e) :: agNativeClosureP.cloneMe(this(), e);
  isNative() :: true;
  getMetaValue() ::
    if(nam.getTxt() = "true",
       true,
       if(nam.getTxt() = "false", false, void));
  print(e) :: printBrackets("native "+nam.getTxt());
  capture() `<- agNative`
};

agNativeClosure(nat, env) :: {
  getNat() :: nat; setNat(f) :: nat := f;
```

```
  getEnv() :: env; setEnv(e) :: env := e;
  cloneMe(f,e) :: {
     c: this().clone(root);
     c.setNat(f); c.setEnv(e);
     c };
  apply(args, e) :: {
             newE : agContext(e.cur, env.ths, e.sup, e.thsActor);
             nat.apply(args, newE) };
`           isClosure() :: true;`
  getMetaValue() :: nat.getMetaValue();
  print(e) :: printBrackets("native closure");
  capture() `<- agClosure`
};

agFunction(nam, pars, body) :: {
  getNam()  :: nam;  setNam(n) :: nam := n;
  getPars() :: pars; setPars(p) :: pars := p;
  getBody() :: body; setBody(b) :: body := b;
  cloneMe(n,p,b) :: {
     c: this().clone(root);
     c.setNam(n); c.setPars(p); c.setBody(b);
     c };
  apply(args, e) :: Error("cannot apply a function directly");
        wrap(e) :: agClosureP.cloneMe(this(), e);
  isFunction() :: true;
  print(e) :: printBrackets("function "+nam.getTxt());
  capture() `<- agFunction`
};

agClosure(fun, env) :: {
  getFun() :: fun; setFun(f) :: fun := f;
  getEnv() :: env; setEnv(e) :: env := e;
  cloneMe(f,e) :: {
     c: this().clone(root);
     c.setFun(f); c.setEnv(e);
     c };
  apply(args, e) :: {
     callframe : env.cur.addFrame();
     if(!args.isTable(),
        { args := args.eval(e);
          if(!args.isTable(),
             Error("Invalid actual arguments: "+args.print(e))) });
     fun.getPars().call(callframe,args,e);
     newE : agContext(callframe, env.ths, env.cur.parent(), e.thsActor);
     fun.getBody().eval(newE) };
  isClosure() :: true;
  print(e) :: printBrackets("closure");
  capture() `<- agClosure`
};

agTable(tbl) :: {
  getTbl()  :: tbl;
  getTable()  :: tbl;
  setTbl(t) :: tbl := t;
  cloneMe(tb) :: {
     c: this().clone(root);
     c.setTbl(tb);
     c };

  call(dct,actuals,e) :: {
```

```
    actT : actuals.getTable();
    forT : this().getTable();
    if (size(actT)!=size(forT),
        Error("non-matching argument list"));
    for(i:1,i<=size(forT),i:=i+1,
        forT[i].bind(dct,actT[i],e))
  };

  get(i) :: tbl[i];
  set(i,v) :: tbl[i] := v;
  isTable() :: true;
  getMetaValue() :: tbl;
  print(e) :: {
    display("[");
    for(i:1, i<=size(tbl), i:=i+1, {
      display(tbl[i].print(e)); if(i<size(tbl), display(", ")) });
    display("]")
  };
  capture() `<- agTable`
};

agText(txt) :: {
  getTxt() :: txt; setTxt(t) :: txt := t;
  cloneMe(tx) :: {
    c: this().clone(root);
    c.setTxt(tx);
    c };
  isText() :: true;
  getMetaValue() :: txt;
  print(e) :: txt;
  capture() `<- agText`
};

agFraction(frc) :: {
  getFrc() :: frc; setFrc(f) :: frc := f;
  cloneMe(fr) :: {
    c: this().clone(root);
    c.setFrc(fr);
    c };
  isFraction() :: true;
  getMetaValue() :: frc;
  print(e) :: frc;
  capture() `<- agFraction`
};

agNumber(num) :: {
  getNum() :: num; setNum(n) :: num := n;
  cloneMe(n) :: {
    c: this().clone(root);
    c.setNum(n);
    c };
  isNumber() :: true;
  getMetaValue() :: num;
  print(e) :: num;
  capture() `<- agNumber`
};

agContext(curDct,thsDct,supDct, thisActor) :: {
  ` if(supDct.isVoid(), Error("..."));`
  cur :: curDct;
```

```
      ths :: thsDct;
      sup :: supDct;
      thsActor :: thisActor;

      print(e) :: printBrackets("context");
      capture() `<- agContext`
    };

    capture() `<- agValue`
  };

  agQuote(exp) :: {
    getExp() :: exp; setExp(e) :: exp := e;
    cloneMe(e) :: {
       c: this().clone(root);
       c.setExp(e);
       c };

    print(e) :: printBrackets("quotation");
    capture() `<- agQuote`
  };

  capture() `<- AbstractGrammar`
};

`prototypes`

agP :: AbstractGrammar();

agValueP :: agP.agValue();
agVoidP :: agValueP.agVoid();
agFractionP :: agValueP.agFraction(0);
agTextP :: agValueP.agText("");
agTableP :: agValueP.agTable([]);
agNumberP :: agValueP.agNumber(0);

agReferenceP :: agP.agReference(agTextP);
agBindingP :: agValueP.agBinding(agTextP, agVoidP, agVoidP);
agObjectP :: agValueP.agObject(agBindingP, agBindingP, agVoidP);

agApplicationP :: agP.agApplication(agReferenceP, agTableP);
agTabulationP :: agP.agTabulation(agReferenceP, agNumberP);
agSyncMessageP :: agP.agSyncMessage(agObjectP, agReferenceP);
agAsyncMessageP :: agP.agAsyncMessage(agObjectP, agReferenceP);
agSuperP :: agP.agSuper(agReferenceP);
agDefinitionP :: agP.agDefinition(agReferenceP, agVoidP);
agDeclarationP :: agP.agDeclaration(agReferenceP, agVoidP);
agAssignmentP :: agP.agAssignment(agReferenceP, agVoidP);
agQuoteP :: agP.agQuote(agVoidP);
agActorP :: agValueP.agActor(agVoidP);
agMailboxP :: agValueP.agMailbox(agVoidP, agVoidP, agVoidP);

agNativeP :: agValueP.agNative(agTextP,void);
agContextP :: agValueP.agContext(agObjectP, agObjectP, agObjectP, agVoidP);
agFunctionP :: agValueP.agFunction(agReferenceP, agTableP, agVoidP);
agClosureP :: agValueP.agClosure(agFunctionP, agContextP);
agNativeClosureP :: agValueP.agNativeClosure(agNativeP, agContextP);
agActorMessageP :: agObjectP.agActorMessage(agContextP);
agActorBehaviorP :: agObjectP.agActorBehavior(agContextP);
```

A.4 Ambient Actor Behavior

```
{
  metaActorBehavior::object({
    actorBehavior  : void;
    mailboxes      : void;
    mbxObservers   : void;
    self           : void;
    context        : void;
    metaSentbox    : void;
    metaInbox      : void;
    metaOutbox     : void;
    metaRcvbox     : void;
    metaRequiredbox: void;
    metaProvidedbox: void;
    metaDisjoinbox : void;
    metaJoinbox    : void;
    currentMessage : void;

    new(anActorBehavior) :: copy({
      actorBehavior:=anActorBehavior;
      mailboxes:=smallmap.new();
      mbxObservers:=smallmap.new().multimapMixin()
    });

    getMailboxes() :: mailboxes;
    getObservers() :: mbxObservers;
    getAddress()   :: self;
    getThisMessage() :: currentMessage;

    getActorBehavior() :: actorBehavior;
    setActorBehavior(anActorBehavior) :: {
      actorBehavior:=anActorBehavior;
      thisActor()#processNextMessage() };

    executeMessage(aMsg, e) :: {
      wrappedMsg: aMsg.agActorMessage(e);
      name: wrappedMsg.getName();
      args: wrappedMsg.getArgs();
      closure: actorBehavior.lookupConstant(name, actorBehavior, e);
      closure.apply(args, e)
    };

    processNextMessage() :: {
      msgToProcessId: metaInbox.findFirst(
                        actorBehavior.containsConstant(
                          el.agActorMessage(context).getName(),
                          actorBehavior,
                          context));
      if(not(is_void(msgToProcessId)), {
         currentMessage:=metaInbox.get(msgToProcessId);
         metaInbox.delete(msgToProcessId);
         actorBehavior.agActorBehavior(context).process(currentMessage);
         thisActor()#processNextMessage()
      })
    };

    initialize(anAgActor) :: {
      self:=anAgActor;
      metaInbox:=agMailboxP.cloneMe("in");
      mailboxes.put("in", metaInbox);
      metaOutbox:=agMailboxP.cloneMe("out");
```

```
  mailboxes.put("out", metaOutbox);
  metaRcvbox:=agMailboxP.cloneMe("rcv");
  mailboxes.put("rcv", metaRcvbox);
  metaSentbox:=agMailboxP.cloneMe("sent");
  mailboxes.put("sent", metaSentbox);
  metaProvidedbox:=agMailboxP.cloneMe("provided");
  mailboxes.put("provided", metaProvidedbox);
  metaRequiredbox:=agMailboxP.cloneMe("required");
  mailboxes.put("required", metaRequiredbox);
  metaJoinbox:=agMailboxP.cloneMe("joined");
  mailboxes.put("joined", metaJoinbox);
  metaDisjoinbox:=agMailboxP.cloneMe("disjoined");
  mailboxes.put("disjoined", metaDisjoinbox);
  context:=agValueP.agContext(actorBehavior,
                              actorBehavior,
                              actorBehavior.getNxt(),
                              this());
  agActorBehaviorP.setContext(context);
  sentbox.addAddObserver(thisActor()#sent);
  metaInbox.addSyncAddObserver({
    thisActor()#processNextMessage()
  });
  metaOutbox.addSyncAddObserver({
    el.agActorMessage(context).getTarget().getAct()#receiveMessage(el)
  });
  metaOutbox.addSyncDeleteObserver({
    msg: el;
    idx: outbox.asVector().findFirst({
      and(el.getName()="receiveMessage", el.getArgs()[1]=msg) });
    if(not(is_void(idx)), outbox.delete(outbox.get(idx)))
  });
  metaRequiredbox.addSyncAddObserver({
    discovery#addRequiredPattern(self, el) });
  metaRequiredbox.addSyncDeleteObserver({
    discovery#removeRequiredPattern(self, el) });
  metaProvidedbox.addSyncAddObserver({
    discovery#addProvidedPattern(self, el) });
  metaProvidedbox.addSyncDeleteObserver({
    discovery#removeProvidedPattern(self, el) });
  metaInbox.add(
    agActorBehaviorP.createMessage(
      self, self, agTextP.cloneMe("init"), agTableP));
  this().processNextMessage();
  `now that we have initialized the actor we can start receiving messages.`
  become(this().receiveMixin())
};

setRootActor(anActor)::{
  rootActor:=anActor
};

joinOn(aPattern, providerActor) :: {
  idx: metaRequiredbox.findFirst(aPattern.getTxt() = el.getTxt());
  if(not(is_void(idx)), {
     metaJoinbox.add(agTableP.cloneMe([providerActor, aPattern])) })
};

disjoinOn(aPattern, providerActor) :: {
  idx: metaJoinbox.findFirst({
    and(and(el.isTable(),
```

```
            el.getTbl()[1].getAct() = providerActor.getAct()),
          el.getTbl()[2].getTxt() = aPattern.getTxt()) });
    if(not(is_void(idx)), {
       resolution: metaJoinbox.delete(idx);
       metaDisjoinbox.add(resolution) })
  };

  receiveMixin() :: {
    receiveMessage(aMsg) :: {
      metaInbox.add(aMsg)
    };
    capture()
  };

  sent(msg) :: {
    if(msg.getName()="receiveMessage", {
       metaMsg: msg.getArgs()[1];
       metaOutbox.remove(metaMsg);
       metaSentbox.add(metaMsg) })
  };


  evaluate(aString) :: {
    msg: agActorBehaviorP.createMessage(
           rootActor,
           rootActor,
           agTextP.cloneMe("evaluate"),
           agTableP.cloneMe([ agTextP.cloneMe(aString) ]));
    receiveMessage(msg)
  }
});

patternInfo(aDevice, aRequirePattern, aProvidePattern,
            aRequireActor, aProviderActor) :: {
  discovery::aDevice;
  requireActor::aRequireActor;
  providerActor::aProviderActor;
  requirePattern::aRequirePattern;
  providePattern::aProvidePattern;

  capture()
};

metaDiscoveryBehavior :: object({
  requiredPatterns   : smallmap.new().multimapMixin();
  providedPatterns   : smallmap.new().multimapMixin();
  `this multimap maps the joined patterns to the actors     `
  matchedPatternActors    : smallmap.new().multimapMixin();
  `this multimap holds maps the discovery actors to the joined `
  `actors so that if a discovery is no longer available then we`
  `can disjoin the joined actors.                           `
  discoveryToJoinedActors    : smallmap.new().multimapMixin();
  `this multimap contains all patterns that were queried for`
  `so that if an actor provides something we can check and  `
  `notify these actors.                                     `
  patternToRequiredDevices: smallmap.new().multimapMixin();

  init() :: {
    provided.add("AmbientTalkVM");
    required.add("AmbientTalkVM");
```

```
  joinBox.addAddObserver(thisActor#joined);
  disjoinBox.addAddObserver(thisActor#disjoined)
};

addRequiredPattern(anActor, aPattern) :: {
  requiredPatterns.put(anActor, aPattern);
  sendRequiredPatternToJoinedDevices(anActor, aPattern)
};

removeRequiredPattern(anActor, aPattern) :: {
  requiredPatterns.deleteItem(anActor, aPattern)
};

addProvidedPattern(anActor, aPattern) :: {
  providedPatterns.put(anActor, aPattern);
  patternToRequiredDevices.iterate({
    if(isMatchingPattern(key, aPattern), value.iterate({
      this().sendMatchingPatternNotification(
        el.discovery, el.requireActor, anActor, key, aPattern)
    }))
  })
};

removeProvidedPattern(anActor, aProvidePattern) :: {
  providedPatterns.deleteItem(anActor, aProvidePattern);
  v: matchedPatternActors.get(aProvidePattern);
  if(not(is_void(v)), v.iterate({
    el.discovery#disjoinNotification(el.requirePattern, anActor)
  }))
};

joined(aResolution) :: {
  sendRequiredPatterns(provider(aResolution))
};

disjoined(aResolution) :: {
  joinedActors: discoveryToJoinedActors.get(provider(aResolution));
  if(not(is_void(joinedActors)), {
    joinedActors.iterate({
      el.requireActor.getAct()#disjoinOn(
        el.requirePattern, el.providerActor)
    });
    discoveryToJoinedActors.delete(provider(aResolution))
  })
};

disjoinNotification(aRequirePattern, anActor) :: {
  v: discoveryToJoinedActors.getKey(customer());
  toDelete: vector.new();
  if(not(is_void(v)), {
    v.iterate({
      if(and(el.requirePattern = aRequirePattern,
             el.providerActor.getAct() = anActor.getAct()),
         {
           toDelete.add(el);
           el.requireActor.getAct()#disjoinOn(
             el.pattern, el.providerActor)
         }) });
    toDelete.iterate(discoveryToJoinedActors.deleteItem(customer(), el))
  })
```

```
};

matchingPatternNotification(
  requireActor, providerActor, aRequirePattern) ::
{
  requireActor.getAct()#joinOn(aRequirePattern, providerActor);
  discoveryToJoinedActors.put(
    customer(),
    root.patternInfo(customer(),
    aRequirePattern,
    void,
    requireActor,
    providerActor))
};

isMatchingPattern(requiredPattern, providedPattern)::{
  and(and(requiredPattern.isText(), providedPattern.isText()),
      requiredPattern.getTxt() = providedPattern.getTxt())
};

sendMatchingPatternNotification(
  aDevice, requireActor, providerActor,
  aRequirePattern, aProvidePattern) ::
{
  aDevice#matchingPatternNotification(
    requireActor, providerActor, aRequirePattern);
  matchedPatternActors.put(
    aProvidePattern,
    root.patternInfo(
      customer(),
      aRequirePattern,
      aProvidePattern,
      requireActor,
      providerActor))
};

sendRequiredPatternToJoinedDevices(requireActor, aPattern) :: {
  joinBox.asVector().iterate({
    provider(el)#queryForRequiredPattern(requireActor, aPattern)
  })
};

sendRequiredPatterns(aRuntime) :: {
  requiredPatterns.iterate({
    value.iterate({
      aRuntime#queryForRequiredPattern(key, el)
    })
  })
};

queryForRequiredPattern(requireActor, aRequirePattern) :: {
  patternToRequiredDevices.put(
    aRequirePattern,
    root.patternInfo(
      customer(),
      aRequirePattern,
      void,
      requireActor,
      void));
  providedPatterns.iterate({
```

```
        value.iterate({
          if(this().isMatchingPattern(aRequirePattern, el), {
             this().sendMatchingPatternNotification(
               customer(),
               requireActor,
               key,
               aRequirePattern,
               el)
          })
        })
      })
    }

  });

  discovery: actor(metaDiscoveryBehavior);

  display("ambient actor model ....... installed", eoln)
}
```

A.5 Native Methods

```
{
  rootObject: agObjectP.clone();

  addNative(anObject, aText, aNativeOp)::{
    nativeText: agTextP.cloneMe(aText);
    anObject.setCst(
      agBindingP.cloneMe(nativeText,
                         agNativeP.cloneMe(aNativeOp, nativeText),
                         anObject.getCst())) };

  addConstant(anObject, aText, aValue)::{
    nativeText: agTextP.cloneMe(aText);
    anObject.setCst(agBindingP.cloneMe(nativeText,
                                       aValue,
                                       anObject.getCst())) };

  native(fun) :: f(args, e) :: {
    i: 0;
    metaArgs: [];
    if(size(args.getTbl()) > 0,
       metaArgs:=tmp[size(args.getTbl())]: void);
    for(i:=1, i<=size(args.getTbl()), i:=i+1,
        metaArgs[i]:=args.getTbl()[i].eval(e).getMetaValue());
    value : fun@metaArgs;
    result: void;
    if(is_number(value), result:=agNumberP.cloneMe(value));
    if(is_fraction(value), result:=agNumberP.cloneMe(value));
    if(is_text(value), result:=agTextP.cloneMe(value));
    if(is_void(value), result:=agVoidP);
    if(is_table(value), result:=agTableP.cloneMe(value));
    if(value~true, result:=_TRUE_);
    if(value~false, result:=_FALSE_);
    result };

  beginNative(args, e) :: {
    i: 0;
    for(i:=1, i<= size(args.getTbl()), i:=i+1,
      args.getTbl()[i].eval(e))
  };

  whileNative(args, e) :: {
    condition: args.getTbl()[1];
    body     : args.getTbl()[2];
    result   : agVoidP;
    loop(aBool)::if(aBool,
                    { result:=body.eval(e);
                      loop(condition.eval(e).getMetaValue()) });
    loop(condition.eval(e).getMetaValue());
    result
  };

  untilNative(args, e) :: {
    condition: args.getTbl()[1];
    body     : args.getTbl()[2];
    result   : agVoidP;
    loop()::{ result:=body.eval(e);
              if(not(condition.eval(e).getMetaValue()), loop()) };
    loop(condition.eval(e).getMetaValue());
    result
  };
```

```
forNative(args, e) :: {
  init     : args.getTbl()[1];
  condition: args.getTbl()[2];
  incr     : args.getTbl()[3];
  body     : args.getTbl()[4];
  result   : agVoidP;
  init.eval(e);
  loop(aBool)::if(aBool,
                  { result:=body.eval(e);
                    incr.eval(e);
                    loop(condition.eval(e).getMetaValue()) });
  loop(condition.eval(e).getMetaValue());
  result
};

tabNative(args, e) :: {
  result[size(args.getTbl())]: void;
  for(i:=1, i<= size(args.getTbl()), i:=i+1,
    result[i]:=args.getTbl()[i].eval(e));
  agTableP.cloneMe(result)
};

ifNative(args, e) :: {
  condition: args.getTbl()[1];
  then     : args.getTbl()[2];
  else     : _VOID_;
  if(size(args.getTbl()) > 2,
     else  := args.getTbl()[3]);
  result   : condition.eval(e);
  if(result~_TRUE_, then.eval(e), else.eval(e))
};

readNative(args, e) :: {
  aString: args.getTbl()[1].eval(e);
  _PARSER_.Read(aString.getTxt())
};

evalNative(args, e) :: {
  anAbstractGrammar: args.getTbl()[1].eval(e);
  anAbstractGrammar.eval(e)
};

captureNative(args, e) :: e.cur;

thisNative(args, e) :: e.ths;

superNative(args, e) :: e.sup;

thisActorNative(args, e) :: { v: e.thsActor; v.getAddress() };

is_numberNative(args, e) ::
  if(args.getTbl()[1].eval(e).isNumber(), _TRUE_, _FALSE_);

is_fractionNative(args, e) ::
  if(args.getTbl()[1].eval(e).isFraction(), _TRUE_, _FALSE_);

is_textNative(args, e) ::
  if(args.getTbl()[1].eval(e).isText(), _TRUE_, _FALSE_);
```

```
is_tableNative(args, e) ::
  if(args.getTbl()[1].eval(e).isTable(), _TRUE_, _FALSE_);

is_voidNative(args, e) ::
  if(args.getTbl()[1].eval(e).isVoid(), _TRUE_, _FALSE_);

is_functionNative(args, e) ::
  if(args.getTbl()[1].eval(e).isFunction(), _TRUE_, _FALSE_);

is_dictionaryNative(args, e) ::
  if(args.getTbl()[1].eval(e).isObject(), _TRUE_, _FALSE_);

is_actorNative(args, e) ::
  if(args.getTbl()[1].eval(e).isActor(), _TRUE_, _FALSE_);

displayNative(args, e) :: {
  for(i: 1, i<=size(args.getTbl()), i:=i+1,
      display(args.getTbl()[i].eval(e).print(e)));
  _VOID_
};


viewNative(args, e) :: {
  newObject: e.cur.addFrame();
  context: agValueP.agContext(newObject,
                              e.ths, e.sup, e.thsActor);
  args.getTbl()[1].eval(context);
  newObject
};

copyNative(args, e) :: {
  if(size(args.getTbl()) > 0, {
     deepcopy: e.cur.picoClone(agVoidP);
     context: agValueP.agContext(deepcopy,
                                 deepcopy.getNxt(),
                                 deepcopy.getNxt().getNxt(),
                                 e.thsActor);
     args.getTbl()[1].eval(context);
     deepcopy.getNxt() },
     e.ths.picoClone(agVoidP))
};

cloneNative(args, e) :: {
  upTo: agVoidP;
  if(size(args.getTbl()) > 0,
     upTo:= args.getTbl()[1].eval(e));
  e.ths.picoClone(upTo)
};

loadNative(args, e) :: {
  filename: args.getTbl()[1].eval(e);
  _PARSER_.Read(readFile(filename.getTxt())).eval(initContext)
};


actorNative(args, e) :: {
  actorBehavior: args.getTbl()[1].eval(e);
  agActorP.cloneMe(actorBehavior)
};
```

```
addToMbxNative(args, e) :: {
  mbxName: args.getTbl()[1].eval(e).getTxt();
  msg    : args.getTbl()[2].eval(e);
  mailbox: e.thsActor.getMailboxes().get(mbxName);
  if(is_void(mailbox), {
     mailbox:=e.thsActor.getMailboxes().put(
                mbxName, agMailboxP.cloneMe(mbxName))});
  mailbox.add(msg);
  _VOID_
};

deleteFromMbxNative(args, e) :: {
  mbxName: args.getTbl()[1].eval(e).getTxt();
  msg    : args.getTbl()[2].eval(e);
  mailbox: e.thsActor.getMailboxes().get(mbxName);
  if(is_void(mailbox),
     _FALSE_,
     if(mailbox.remove(msg), _TRUE_, _FALSE_))
};

messagesNative(args, e) :: {
  mbxName: args.getTbl()[1].eval(e).getTxt();
  mailbox: e.thsActor.getMailboxes().get(mbxName);
  if(is_void(mailbox),
     agTableP,
     agTableP.cloneMe(mailbox.getMetaValue().asTable()))
};

executeNative(args, e) :: {
  msg: args.getTbl()[1].eval(e);
  e.thsActor.executeMessage(msg, e)
};

becomeNative(args, e) :: {
  newBehavior  : args.getTbl()[1].eval(e);
  e.thsActor.setBehavior(newBehavior);
  _VOID_
};

thisMessageNative(args, e) :: {
  e.thsActor.getThisMessage()
};

addAddObserverNative(args, e) :: {
  mbxName  : args.getTbl()[1].eval(e);
  notifyMsg: args.getTbl()[2].eval(e);
  mbx     : e.thsActor.getMailboxes().get(mbxName.getTxt());
  inbox   : e.thsActor.getMailboxes().get("in");
  notify(observedMsg) :: {
    v: e.thsActor.getObservers().get(mbxName.getTxt());
    if(and(not(is_void(v)),
           not(v.detect({
                 and(observedMsg.isObject(),
                     el[1].getName().getTxt() =
                     observedMsg.agActorMessage(e)
                       .getName().getTxt())
               }))), {
      msgToPost: notifyMsg.picoClone(agVoidP);
      reifyMsg: msgToPost.agActorMessage(e);
      reifyMsg.setArgs(agTableP.cloneMe([ observedMsg ]));
```

```
        target: reifyMsg.getTarget().getAct();
        if(target = e.thsActor.getAddress().getAct(),
           { inbox.add(msgToPost) },
           { target#receiveMessage(msgToPost) }) })
  };
  e.thsActor.getObservers().put(
    mbxName.getTxt(), [notifyMsg.agActorMessage(e), notify]);
  mbx.addSyncAddObserver(notify(el));
  _VOID_
};

addNative(rootObject, "+", native(+));
addNative(rootObject, "-", native(-));
addNative(rootObject, "*", native(*));
addNative(rootObject, "/", native(/));
addNative(rootObject, "^", native(^));
addNative(rootObject, "//", native(//));
addNative(rootObject, "<", native(<));
addNative(rootObject, "<=", native(<=));
addNative(rootObject, ">", native(>));
addNative(rootObject, ">=", native(>=));
addNative(rootObject, "=", native(=));
addNative(rootObject, "!=", native(!=));
addNative(rootObject, "~", native(~));
addNative(rootObject, "!~", native(!~));
addNative(rootObject, "&", native(&));
addNative(rootObject, "|", native(|));
addNative(rootObject, "!", native(!));
addNative(rootObject, "and", native(and));
addNative(rootObject, "or", native(or));
addNative(rootObject, "not", native(not));
addNative(rootObject, "begin", beginNative);
addNative(rootObject, "while", whileNative);
addNative(rootObject, "until", untilNative);
addNative(rootObject, "for", forNative);
addNative(rootObject, "tab", tabNative);
addNative(rootObject, "size", native(size));
addNative(rootObject, "if", ifNative);
addNative(rootObject, "display", displayNative);
addNative(rootObject, "accept", native(accept));
addNative(rootObject, "length", native(length));
addNative(rootObject, "explode", void);
addNative(rootObject, "implode", void);
addNative(rootObject, "this", thisNative);
addNative(rootObject, "super", superNative);
addNative(rootObject, "clone", cloneNative);
addNative(rootObject, "equivalent", native(equivalent));
addNative(rootObject, "abs", native(abs));
addNative(rootObject, "trunc", native(trunc));
addNative(rootObject, "char", void);
addNative(rootObject, "ord", void);
addNative(rootObject, "number", native(number));
addNative(rootObject, "text", native(text));
addNative(rootObject, "random", native(random));
addNative(rootObject, "sin", native(sin));
addNative(rootObject, "cos", native(cos));
addNative(rootObject, "tangent", native(tangent));
addNative(rootObject, "sqrt", native(sqrt));
addNative(rootObject, "exp", native(exp));
addNative(rootObject, "log", native(log));
```

```
addNative(rootObject, "arcsin", native(arcsin));
addNative(rootObject, "arccos", native(arccos));
addNative(rootObject, "arctan", native(arctan));
addNative(rootObject, "call", void);
addNative(rootObject, "continue", void);
addNative(rootObject, "capture", captureNative);
addNative(rootObject, "is_void", is_voidNative);
addNative(rootObject, "is_number", is_numberNative);
addNative(rootObject, "is_fraction", is_fractionNative);
addNative(rootObject, "is_text", is_textNative);
addNative(rootObject, "is_table", is_tableNative);
addNative(rootObject, "is_function", is_functionNative);
addNative(rootObject, "is_dictionary", is_dictionaryNative);
addNative(rootObject, "is_environment", void);
addNative(rootObject, "is_continuation", void);
addNative(rootObject, "read", readNative);
addNative(rootObject, "eval", evalNative);
addNative(rootObject, "def", void);
addNative(rootObject, "dcl", void);
addNative(rootObject, "set", void);
addNative(rootObject, "send", void);
addNative(rootObject, "super_send", void);
addNative(rootObject, "time", native(time));
addNative(rootObject, "load", loadNative);
addNative(rootObject, "inspect", native(inspect));
addNative(rootObject, "error", native(error));
addNative(rootObject, "doesNotUnderstand", void);
addNative(rootObject, "readFile", native(readFile));
addNative(rootObject, "table", native(tab));

addNative(rootObject, "execute", executeNative);
addNative(rootObject, "copy", copyNative);
addNative(rootObject, "object", viewNative);
addNative(rootObject, "view", viewNative);
addNative(rootObject, "sleep", void);
addNative(rootObject, "setTimeout", void);
addNative(rootObject, "startNetwork", native(startNetwork));
addNative(rootObject, "stopNetwork", native(stopNetwork));
addNative(rootObject, "actor", actorNative);
addNative(rootObject, "thisActor", thisActorNative);
addNative(rootObject, "become", becomeNative);
addNative(rootObject, "is_actor", is_actorNative);
addNative(rootObject, "messages", messagesNative);
addNative(rootObject, "add", addToMbxNative);
addNative(rootObject, "delete", deleteFromMbxNative);
addNative(rootObject, "thisMessage", thisMessageNative);
addNative(rootObject, "uid", native(uid));
addNative(rootObject, "addAddObserver", addAddObserverNative);
addNative(rootObject, "addDeleteObserver", void);
addNative(rootObject, "removeAddObserver", void);
addNative(rootObject, "removeDeleteObserver", void);

addConstant(rootObject, "true", _TRUE_);
addConstant(rootObject, "false", _FALSE_);
addConstant(rootObject, "void", _VOID_);
addConstant(rootObject, "eoln", _EOLN_);

initContext: agValueP.agContext(rootObject,
                                rootObject,
                                rootObject.getNxt(),
```

```
                                    agVoidP);

    display("loading init.pco ...", eoln);
    _PARSER_.Read(readFile("init.pco")).eval(initContext);

    addConstant(rootObject, "root", rootObject);

    agActorBehaviorP.setContext(initContext);

    rootActor: void;
    rootActor:= agActorP.cloneMe(rootObject);
    evaluator: rootActor.getAct();
    evaluator#setRootActor(rootActor);

    display("natives ....... installed", eoln)
  }
```

Appendix B

Code Listing of BlueChat

Color	Code deals with...
yellow	Concurrency
green	Ambient Resources
turquoise	Communication
purple	Application
red	Volatile Connections

Table B.1: Legend

B.1 NETLayer

```
package btchat;

import javax.microedition.io.*;
import javax.bluetooth.*;
import java.io.*;
import java.util.*;
import javax.bluetooth.RemoteDevice;
import javax.bluetooth.DeviceClass;
import javax.bluetooth.ServiceRecord;

/**
 * BlueChat example application.
 * Originally published in Java Developer's Journal (volume 9 issue 2).
 * Updated by Ben Hui on www.benhui.net.
 * Copyright: (c) 2003-2004
 * Author: Ben Hui
 *
 * YOU ARE ALLOWED TO USE THIS CODE FOR EDUCATIONAL, PERSONAL TRAINNING,
 * REFERENCE PURPOSE. YOU MAY DISTRIBUTE THIS CODE AS-IS OR MODIFIED FORM.
 * HOWEVER, YOU CANNOT USE THIS CODE FOR COMMERCIAL PURPOSE. THIS INCLUDE,
 * BUT NOT LIMITED TO, PRODUCING COMMERCIAL SOFTWARE, CONSULTANT SERVICE,
 * PROFESSIONAL TRAINNING MATERIAL.
 *
 * This is the main class for handling bluetooth connectivity and
 * device/service discovery process. This class does many things, including
 * - search for bluetooth devices (query())
 * - create a local BlueChat server and register it with bluetooth (run())
 * - search for remote BlueChat services using searchServices()
 * - handle incoming connection request from remote BlueChat
 * - establish connection to remote BlueChat
 *
 * @author Ben Hui
 * @version 1.0
 */
public class NetLayer implements Runnable
{
  public final static int SIGNAL_HANDSHAKE = 0;
  public final static int SIGNAL_MESSAGE = 1;
  public final static int SIGNAL_TERMINATE = 3;
  public final static int SIGNAL_HANDSHAKE_ACK = 4;
  public final static int SIGNAL_TERMINATE_ACK = 5;
  private final static UUID uuid = new UUID("102030405060708090A0B0C0D0E0F010",
false);
  private final static int SERVICE_TELEPHONY = 0x400000;
  LocalDevice localDevice = null;
  DiscoveryAgent agent = null;
  StreamConnectionNotifier server;
  BTListener callback = null;
  boolean done = false;
  String localName = "";
  Vector endPoints = new Vector();
  Vector pendingEndPoints = new Vector();
  Hashtable serviceRecordToEndPoint = new Hashtable();
  Object lock = new Object();
  Timer timer = new Timer();
```

```
  public NetLayer()
  {
  }
```

```
  public void init(String name, BTListener callback)
  {
    try {
      this.localName = name;
      this.callback = callback;
      localDevice = LocalDevice.getLocalDevice(); // obtain reference to sin-
gleton
      localDevice.setDiscoverable(DiscoveryAgent.GIAC); // set Discover mode to
GIAC
      agent = localDevice.getDiscoveryAgent(); // obtain reference to singleton
      Util.printLocalDevice( localDevice );
      Thread thread = new Thread( this );
      thread.start();
    }
    catch (BluetoothStateException e) {
      e.printStackTrace();
    }
    catch (IOException e) {
      e.printStackTrace();
    }
  }
```

```
  public void disconnect()
  {
    done = true;
    try {
      server.close();
    }
    catch (IOException ex) {
    }

    for ( int i=0; i < endPoints.size(); i++ )
    {
      EndPoint endpt = (EndPoint) endPoints.elementAt( i );
      endpt.putString( NetLayer.SIGNAL_TERMINATE, "end" );
      endpt.sender.stop();
      endpt.reader.stop();
    }
  }
```

```
  public void query()
  {
    try {
      agent.startInquiry(DiscoveryAgent.GIAC, new Listener());
    }
    catch (BluetoothStateException e)
    {
      e.printStackTrace();
      log(e.getClass().getName()+" "+e.getMessage());
    }
```

```
  }

  public EndPoint findEndPointByRemoteDevice( RemoteDevice rdev )
  {
    for ( int i=0; i < endPoints.size(); i++ )
    {
      EndPoint endpt = (EndPoint) endPoints.elementAt( i );
      if ( endpt.remoteDev.equals( rdev ) )
      {
        return endpt;
      }
    }
    return null;
  }

  public EndPoint findEndPointByTransId( int id )
  {
    for ( int i=0; i < pendingEndPoints.size(); i++ )
    {
      EndPoint endpt = (EndPoint) pendingEndPoints.elementAt( i );
      if ( endpt.transId == id )
      {
        return endpt;
      }
    }
    return null;
  }

  public void sendString( String s )
  {
    for ( int i=0; i < endPoints.size(); i++ )
    {
      EndPoint endpt = (EndPoint) endPoints.elementAt( i );
      endpt.putString( NetLayer.SIGNAL_MESSAGE, s );
    }
  }

  public void cleanupRemoteEndPoint( EndPoint endpt )
  {
    endpt.reader.stop();
    endpt.sender.stop();
    endPoints.removeElement( endpt );
  }

  public void run()
  {
    StreamConnection c = null;
    try
    {
      server =  (StreamConnectionNotifier)Connector.open(
          "btspp://localhost:" + uuid.toString() +";name=BlueChatApp");

      ServiceRecord rec = localDevice.getRecord( server );
```

```
      rec.setAttributeValue( 0x0008, new DataElement( DataElement.U_INT_1, 0xFF
) );
      Util.printServiceRecord( rec );
      rec.setDeviceServiceClasses(
          SERVICE_TELEPHONY  );
    } catch (Exception e)
    {
      e.printStackTrace();
      log(e.getClass().getName()+" "+e.getMessage());
    }
```

```
    while( !done)
    {
      try {
        ChatMain.instance.gui_log( "", "Ready to accept connection. Wait..." );
        c = server.acceptAndOpen();

        RemoteDevice rdev = RemoteDevice.getRemoteDevice( c );
        EndPoint endpt = findEndPointByRemoteDevice( rdev );
        if ( endpt != null )
        {
        } else
        {
          endpt = new EndPoint( this, rdev, c);
          Thread t1 = new Thread( endpt.sender );
          t1.start();
          Thread t2 = new Thread( endpt.reader );
          t2.start();
          endPoints.addElement( endpt );
        }
      }
      catch (IOException e) {
        e.printStackTrace();
        log(e.getClass().getName()+" "+e.getMessage());
        if (c != null)
          try {
            c.close();
          }
          catch (IOException e2) {
            // ignore
          }
      }
      finally {
        // nothing to do here
      }
    } // while !done
  } // end run()

  public static void log( String s)
  {
    if ( ChatMain.isDebug )
      ChatMain.instance.gui_log( "N", s );
  }

  class Listener implements DiscoveryListener
```

```
{
  public void deviceDiscovered(RemoteDevice remoteDevice,
                               DeviceClass deviceClass)
  {
    try {
    }
    catch (IOException ex) {
    }
      try
      {
        EndPoint endpt = new EndPoint(NetLayer.this, remoteDevice, null);
        pendingEndPoints.addElement( endpt );
      } catch (Exception e)
      {
        e.printStackTrace();
        log(e.getClass().getName()+" "+e.getMessage());

      }
  }
```

```
  public void inquiryCompleted(int transId)
  {
    timer.schedule( new DoServiceDiscovery(), 100 );
  }
```

```
  public void servicesDiscovered(int transId, ServiceRecord[] svcRec)
  {
    try {
      for ( int i=0; i< svcRec.length; i++ )
      {
        Util.printServiceRecord( svcRec[i] );
        EndPoint endpt = findEndPointByTransId( transId );
        serviceRecordToEndPoint.put( svcRec[i], endpt );
      }
    }
    catch (Exception e) {
      e.printStackTrace();
      log(e.getClass().getName());
      log(e.getMessage());
    }
  }
```

```
  public void serviceSearchCompleted(int transID, int respCode)
  {
    log("invoke serviceSearchCompleted: "+transID);
    if ( respCode == SERVICE_SEARCH_COMPLETED )
      log("SERVICE_SEARCH_COMPLETED");
    else if ( respCode == SERVICE_SEARCH_TERMINATED )
      log("SERVICE_SEARCH_TERMINATED");
    else if ( respCode == SERVICE_SEARCH_ERROR )
      log("SERVICE_SEARCH_ERROR");
    else if ( respCode == SERVICE_SEARCH_NO_RECORDS )
      log("SERVICE_SEARCH_NO_RECORDS");
    else if ( respCode == SERVICE_SEARCH_DEVICE_NOT_REACHABLE )
      log("SERVICE_SEARCH_DEVICE_NOT_REACHABLE");
```

```
      for ( Enumeration records = serviceRecordToEndPoint.keys(); re-
cords.hasMoreElements(); )
      {
        try {
        ServiceRecord rec = (ServiceRecord) records.nextElement();
        String url  = rec.getConnectionURL( ServiceRe-
cord.NOAUTHENTICATE_NOENCRYPT, false );
        StreamConnection con = (StreamConnection)Connector.open( url );
        EndPoint endpt = (EndPoint) serviceRecordToEndPoint.get( rec );
        if ( endpt != null )
        {
          endpt.con = con;
          Thread t1 = new Thread( endpt.sender );
          t1.start();
          Thread t2 = new Thread( endpt.reader );
          t2.start();
          endPoints.addElement( endpt );
          endpt.putString( NetLayer.SIGNAL_HANDSHAKE, localName );
        }
        } catch (Exception e)
        {
          e.printStackTrace();
        }
      } // for
      serviceRecordToEndPoint.clear();
      synchronized( lock )
      {
        lock.notifyAll();
      }
    }
  } // inner class Listener
```

```
  class DoServiceDiscovery extends TimerTask
  {
    public void run()
    {
      for (int i = 0; i < pendingEndPoints.size(); i++)
      {
        EndPoint endpt = (EndPoint) pendingEndPoints.elementAt(i);
        try {
          endpt.transId = agent.searchServices(null,
                                               new UUID[] { uuid },
                                               endpt.remoteDev,
                                               new Listener());
          synchronized( lock )
          {
            try {
              lock.wait();
            }
            catch (InterruptedException ex) {
            }
          }
        }
        catch (BluetoothStateException e) {
```

```
          e.printStackTrace();
          log(e.getClass().getName()+" "+e.getMessage());

        }
      } // for
      pendingEndPoints.removeAllElements();
      ChatMain.instance.gui_log( "", "You can start chatting now" );
    }
  }
}
```

B.2 EndPoint

```
package btchat;
import javax.bluetooth.*;
import javax.microedition.io.*;
import java.io.*;
import java.util.*;


/**
 * BlueChat example application.
 * Originally published in Java Developer's Journal (volume 9 issue 2).
 * Updated by Ben Hui on www.benhui.net.
 * Copyright: (c) 2003-2004
 * Author: Ben Hui
 *
 * YOU ARE ALLOWED TO USE THIS CODE FOR EDUCATIONAL, PERSONAL TRAINNING,
 * REFERENCE PURPOSE. YOU MAY DISTRIBUTE THIS CODE AS-IS OR MODIFIED FORM.
 * HOWEVER, YOU CANNOT USE THIS CODE FOR COMMERCIAL PURPOSE. THIS INCLUDE,
 * BUT NOT LIMITED TO, PRODUCING COMMERCIAL SOFTWARE, CONSULTANT SERVICE,
 * PROFESSIONAL TRAINNING MATERIAL.
 *
 *
 * A EndPoint object represent all the connection attribute of an active
BlueChat node.
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2003</p>
 * @author Ben Hui
 * @version 1.0
 */
public class EndPoint
{
  // remote device object
  RemoteDevice remoteDev;
  DeviceClass remoteClass;
  String remoteUrl;
  StreamConnection con;
  int transId = -1; // -1 must be used for default. cannot use 0

  Sender sender;
  Reader reader;

  String localName;
  String remoteName;

  BTListener callback;
  NetLayer btnet;
  Vector msgs = new Vector();

  public EndPoint( NetLayer btnet, RemoteDevice rdev, StreamConnection c )
  {
    this.btnet = btnet;
    remoteDev = rdev;
    try {
      remoteName = rdev.getFriendlyName(false); // this is a temp name
    }
    catch (IOException ex) {
```

```
      remoteName = "Unknown";
    }
    localName = btnet.localName;
    callback = btnet.callback;
    con = c;
    sender = new Sender();
    sender.endpt = this;
    reader = new Reader();
    reader.endpt = this;
  }

  public synchronized void putString( int signal, String s )
  {
    msgs.addElement( new ChatPacket( signal, s ) );
    synchronized( sender )
    {
      sender.notify();
    }
  }

  public synchronized ChatPacket getString()
  {
    if ( msgs.size() > 0 )
    {
      ChatPacket s = (ChatPacket) msgs.firstElement();
      msgs.removeElementAt(0);
      return s;
    } else
    {
      return null;
    }
  }

  public synchronized boolean peekString()
  {
    return ( msgs.size() > 0 );
  }

  private static void log( String s )
  {
    System.out.println("EndPoint: "+s);
  }
}
```

B.3 Sender

```
package btchat;

/**
 * BlueChat example application.
 * Originally published in Java Developer's Journal (volume 9 issue 2).
 * Updated by Ben Hui on www.benhui.net.
 * Copyright: (c) 2003-2004
 * Author: Ben Hui
 *
 * YOU ARE ALLOWED TO USE THIS CODE FOR EDUCATIONAL, PERSONAL TRAINNING,
 * REFERENCE PURPOSE. YOU MAY DISTRIBUTE THIS CODE AS-IS OR MODIFIED FORM.
 * HOWEVER, YOU CANNOT USE THIS CODE FOR COMMERCIAL PURPOSE. THIS INCLUDE,
 * BUT NOT LIMITED TO, PRODUCING COMMERCIAL SOFTWARE, CONSULTANT SERVICE,
 * PROFESSIONAL TRAINNING MATERIAL.
 *
 * Sender thread that send out signal and data to a bluetooth connection.
 * <p>Description: Sender is a Runnable implementation that send signal and
data (String)
 *  to connected DataInputStream. Each EndPoint has it own sender thread.</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * @author Ben Hui
 * @version 1.0
 */
import java.io.*;

public class Sender implements Runnable
{
  public EndPoint endpt;
  private boolean done = false;
  public Sender()
  {
  }

  public void stop()
  {
    done = true;
  }

  public void run()
  {
    try
    {
      DataOutputStream dataout = endpt.con.openDataOutputStream();
      while( !done )
      {
        if ( ! endpt.peekString()  )
        {
          synchronized (this) {
            this.wait(5000);
          }
        }
        ChatPacket s = endpt.getString();
        if ( s != null )
        {
          dataout.writeInt(s.signal);
```

```
          dataout.writeUTF(s.msg );
          dataout.flush();
        }

        if ( s != null && s.signal == NetLayer.SIGNAL_TERMINATE )
        {
          stop();
        }
      } // while !done
      dataout.close();
    } catch (Exception e)
    {
      e.printStackTrace();
      log(e.getClass().getName()+"  "+e.getMessage());
    }
  }
  private static void log( String s)
  {
    System.out.println("Sender:  "+s);
    if ( ChatMain.isDebug )
      ChatMain.instance.gui_log( "S", s );
  }
}
```

B.4 Reader

```
package btchat;

/**
 * BlueChat example application.
 * Originally published in Java Developer's Journal (volume 9 issue 2).
 * Updated by Ben Hui on www.benhui.net.
 * Copyright: (c) 2003-2004
 * Author: Ben Hui
 *
 * YOU ARE ALLOWED TO USE THIS CODE FOR EDUCATIONAL, PERSONAL TRAINNING,
 * REFERENCE PURPOSE. YOU MAY DISTRIBUTE THIS CODE AS-IS OR MODIFIED FORM.
 * HOWEVER, YOU CANNOT USE THIS CODE FOR COMMERCIAL PURPOSE. THIS INCLUDE,
 * BUT NOT LIMITED TO, PRODUCING COMMERCIAL SOFTWARE, CONSULTANT SERVICE,
 * PROFESSIONAL TRAINNING MATERIAL.
 *
 * Reader thread that read in signal and data from a bluetooth connection.
 * <p>Description: Reader is a Runnable implementation that read in signal and
data (String)
 *  from connected DataInputStream. Each EndPoint has it own reader thread.</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * @author Ben Hui
 * @version 1.0
 */
import java.io.*;

public class Reader implements Runnable
{
  public EndPoint endpt;
  private boolean done = false;
  public Reader() {
  }
  public void stop()
  {
    done = true;
  }

  public void run()
  {
    try
    {
      DataInputStream datain = endpt.con.openDataInputStream();
      while ( !done )
      {
        int signal = datain.readInt();
        if ( signal == NetLayer.SIGNAL_MESSAGE )
        {
          String s = datain.readUTF();
          ChatPacket packet = new ChatPacket( NetLayer.SIGNAL_MESSAGE,
endpt.remoteName, s );
          endpt.callback.handleAction( BTListener.EVENT_RECEIVED, endpt, packet
);
        } else if ( signal == NetLayer.SIGNAL_HANDSHAKE )
        {
          String s = datain.readUTF();
          endpt.remoteName = s;
```

```
          endpt.putString( NetLayer.SIGNAL_HANDSHAKE_ACK, endpt.localName );
          endpt.callback.handleAction( BTListener.EVENT_JOIN, endpt, null );
        } else if ( signal == NetLayer.SIGNAL_TERMINATE )
        {
          endpt.putString( NetLayer.SIGNAL_TERMINATE_ACK, "end" );
          endpt.callback.handleAction( BTListener.EVENT_LEAVE, endpt, null );
          endpt.btnet.cleanupRemoteEndPoint( endpt );
          stop();
        } else if ( signal == NetLayer.SIGNAL_HANDSHAKE_ACK )
        {
          String s = datain.readUTF();
          endpt.remoteName = s;
        } else if ( signal == NetLayer.SIGNAL_TERMINATE_ACK )
        {
          System.out.println("read in TERMINATE_ACK from "+endpt.remoteName);
        }
      } // while !done

      datain.close();
    } catch (Exception e)
    {
      e.printStackTrace();
      log(e.getClass().getName()+"  "+e.getMessage());
    }
  }
  private static void log( String s)
  {
    System.out.println("Reader: "+s);
    if ( ChatMain.isDebug )
      ChatMain.instance.gui_log( "R", s );
  }
}
```

B.5 ChatPacket

```
package btchat;

/**
 * BlueChat example application.
 * Originally published in Java Developer's Journal (volume 9 issue 2).
 * Updated by Ben Hui on www.benhui.net.
 * Copyright: (c) 2003-2004
 * Author: Ben Hui
 *
 * YOU ARE ALLOWED TO USE THIS CODE FOR EDUCATIONAL, PERSONAL TRAINNING,
 * REFERENCE PURPOSE. YOU MAY DISTRIBUTE THIS CODE AS-IS OR MODIFIED FORM.
 * HOWEVER, YOU CANNOT USE THIS CODE FOR COMMERCIAL PURPOSE. THIS INCLUDE,
 * BUT NOT LIMITED TO, PRODUCING COMMERCIAL SOFTWARE, CONSULTANT SERVICE,
 * PROFESSIONAL TRAINNING MATERIAL.
 *
 * A holder object for BlueChat network packet data.
 * <p>Description: ChatPacket can represent severl type of message, which is
defined
 * by NetLayer.SIGNAL_XXX enumeration. The common type is SIGNAL_MESSAGE, which
 * hold an user entered message to sent across the virtual chat room.</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * @author Ben Hui
 * @version 1.0
 */

public class ChatPacket
{
  // signal, must be one of NetLayer.SIGNAL_XXX
  public int signal;
  // indicate the nick name of the sender
  public String sender;
  // the message content
  public String msg;

  public ChatPacket(int signal, String msg)
  {
    this.signal = signal;
    this.msg = msg;
  }

  public ChatPacket(int signal, String sender, String msg)
  {
    this.signal = signal;
    this.sender = sender;
    this.msg = msg;
  }

  public ChatPacket()
  {
  }

}
```

B.6 BTListener

```
package btchat;

/**
 * BlueChat example application.
 * Originally published in Java Developer's Journal (volume 9 issue 2).
 * Updated by Ben Hui on www.benhui.net.
 * Copyright: (c) 2003-2004
 * Author: Ben Hui
 *
 * YOU ARE ALLOWED TO USE THIS CODE FOR EDUCATIONAL, PERSONAL TRAINNING,
 * REFERENCE PURPOSE. YOU MAY DISTRIBUTE THIS CODE AS-IS OR MODIFIED FORM.
 * HOWEVER, YOU CANNOT USE THIS CODE FOR COMMERCIAL PURPOSE. THIS INCLUDE,
 * BUT NOT LIMITED TO, PRODUCING COMMERCIAL SOFTWARE, CONSULTANT SERVICE,
 * PROFESSIONAL TRAINNING MATERIAL.
 *
 * Interface for BlueChat NetLayer callback.
 * <p>Description: Implementation of this interface will handle BlueChat net-
work event.</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * @author Ben Hui
 * @version 1.0
 */
public interface BTListener
{
  public final static String EVENT_JOIN = "join";
  public final static String EVENT_LEAVE = "leave";
  public final static String EVENT_RECEIVED = "received";
  public final static String EVENT_SENT = "sent";
  public void handleAction( String action, Object param1, Object param2 );
}
```

Appendix C

Code Listing of AmbientChat

Color	Code deals with...
yellow	Concurrency
green	Ambient Resources
turquoise	Communication
purple	Application
red	Volatile Connections

Table C.1: Legend

C.1 Ambient Sensor

```
ambientSensorBehaviour :: object({
  pattern : void;
  onJoin : void;
  onDisjoin : void;
  new(p,onJoinMsg,onDisjoinMsg) :: copy({
      pattern := p;
      onJoin := onJoinMsg;
      onDisjoin := onDisjoinMsg
  });

  init() :: {
      required.add(pattern);
      joinBox.addAddObserver(thisActor()#joined);
      disjoinBox.addAddObserver(thisActor()#disjoined)
  };

  joined(resolution) :: {
      copy: onJoin.copy();
      copy.setArgs([ provider(resolution) ]);
      outbox.add(copy)
  };

  disjoined(resolution) :: {
      disjoinBox.delete(resolution);
      copy: onDisjoin.copy();
      copy.setArgs([ provider(resolution) ]);
      outbox.add(copy)
  }
});

AmbientSensor(pattern,onJoin,onDisjoin) ::
  actor(ambientSensorBehaviour.new(pattern,onJoin,onDisjoin));
```

C.2 AmbientChat

```
IMBehaviour :: root.extend({
  uid       : "anonymous";
  buddies   : void;
  online    : void;
  imSensor  : void;

  new(id) :: copy({
    uid       := id;
    buddies   := smallmap.new();
    online    := smallmap.newWithComparator(key1~key2)
  });

  init() :: {
    imSensor := AmbientSensor(IMPATTERN, thisActor()#onChatJoined, thisAc-
tor()#onChatLeft);
    publish(IMPATTERN);
    publish(uid);
    stdio#display("Chat started as ",uid, eoln)
  };

  getUID() :: { uid };
  whoIsOnline() :: { online.getValuesVector().iterate(stdio#display(el," ")) };
  addBuddy(buddyId) :: { buddies.put(buddyId, WeakMonoAmbientRef(buddyId)) };
  sendMessageTo(buddyId,text) :: {
     if (buddies.containsKey(buddyId),
         when( buddies.get(buddyId)#receive(uid,text), {
               receive(uid, text)
         }),
         { stdio#display(buddyId, " added to buddylist",eoln);
           addBuddy(buddyId);
           sendMessageTo(buddyId, text) })
  };
  receive(from, text) :: { stdio#display(from,": ",text,eoln) };

  onChatJoined(buddy) :: {
     stdio#display("InstantMessenger detected in the ambient: ",buddy,eoln);
     when (buddy#getUID(), {
        stdio#display("buddy online: ",content,eoln);
        online.put(buddy, content)
     })
  };

  onChatLeft(buddy) :: {
     if (online.containsKey(buddy),
         stdio#display("buddy offline: ",online.delete(buddy),eoln))
  }

}).futuresMixin();

IMActor(name) :: actor(IMBehaviour.new(name));
```

Bibliography

[AC93] Gul Agha and Christian J. Callsen. Actorspace: an open distributed programming paradigm. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 23–32. ACM Press, 1993.

[ACG00] Isabelle Attali, Denis Caromel, and Romain Guider. A step toward automatic distribution of java programs. In *Fourth International Conference on Formal methods for open object-based distributed systems IV*, pages 141–161, Norwell, MA, USA, 2000. Kluwer Academic Publishers.

[Agh86] G. Agha. *Actors—A Model of Concurrent Computation for Distributed Systems.* MIT Press, 1986.

[Agh90] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, 1990.

[AH88] Gul Agha and Carl Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, Computer Systems Series, pages 37–53. The MIT Press: Cambridge, MA, USA, 1988.

[AMST97] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[AS96] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, USA, 1996.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[Ben86] Jon Bentley. Programming pearls: little languages. *Commun. ACM*, 29(8):711–721, 1986.

[BGL98] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Löhr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[BI93] Andrew P. Black and Mark P. Immel. Encapsulating plurality. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 57–79, London, UK, 1993. Springer-Verlag.

[BNOW93] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 217–230, New York, NY, USA, 1993. ACM Press.

[BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Comput. Surv.*, 21(3):261–322, 1989.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[BY87] Jean-Pierre Briot and Akinori Yonezawa. Inheritance and synchronization in concurrent oop. In *European conference on object-oriented programming on ECOOP '87*, pages 32–40. Springer-Verlag, 1987.

[Car89] Denis Caromel. Service, asynchrony, and wait-by-necessity. *Journal of Object-Oriented Programming*, pages 12–22, November/December 1989.

[Car95] Luca Cardelli. A language with distributed scope. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, pages 286–297, New York, NY, 1995.

[CCW03] M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Trans. Software Engineering*, 29(12):1059–1071, dec 2003.

[CDK05] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems (4th ed.): concepts and design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[CH05] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects.* Springer Verlag, 2005.

[CHS04] Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*, pages 123–134. ACM Press, 2004.

[CJ02] Gianpaolo Cugola and H.-Arno Jacobsen. Using publish/subscribe middleware for mobile systems. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):25–33, 2002.

[CM93] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 482–501. Springer-Verlag, 1993.

[CMP05] Gianpaolo Cugola, Amy L. Murphy, and Gian Pietro Picco. Content-based publish-subscribe in a mobile environment. In A. Corradi and P. Bellavista, editors, *Mobile Middleware*. CRC Press, 2005.

[CNP00] G. Cugola, E. Di Nitto, and G. P. Pico. Content-based dispatching in a mobile environment. In *Proceedings of WSDAAL*. ACM Press, 2000.

[CSM+06] Pascal Cherrier, Daniel Stern, Holger Mügge, Wolfgang De Meuter, and Eric Tanter. 1st international workshop on software engineering of pervasive services, June 2006.

[CUL89] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 49–70, New York, NY, USA, 1989. ACM Press.

[DD03] Theo D'Hondt and Wolfgang De Meuter. Of first-class methods and dynamic scope. In *Proceedings of LMO Conference, RSTI - L'objet*, pages 137–149, Cachan, France, 2003. RSTI.

[DDD03] Wolfgang De Meuter, Theo D'Hondt, and Jessie Dedecker. Intersecting classes and prototypes. In Manfred Broy and Alexandre V. Zamulin, editors, *Ershov Memorial Conference*, volume 2890 of *Lecture Notes in Computer Science*. Springer, 2003. Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003.

[DDD04] Theo D'Hondt, Wolfgang De Meuter, and Jessie Dedecker. Pico: Scheme for mere mortals. In *1st European Lisp and Scheme Workshop, Oslo, Norway*, 2004.

[De 04] Wolfgang De Meuter. *Move Considered Harmful: A Language Design Approach to Mobility and Distribution for Open Networks*. PhD thesis, Vrije Universiteit Brussel, September 2004.

[DFWB98] Nigel Davies, Adrian Friday, Stephen P. Wade, and Gordon S. Blair. An asynchronous distributed systems platform for heterogeneous environments. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 66–73. ACM Press, 1998.

[Dic92] Kenneth Dickey. Scheming with objects. *AI Expert*, 7(10):24–33, October 1992.

[DTM+05] Wolfgang De Meuter, Eric Tanter, Stijn Mostinckx, Tom Van Cutsem, and Jessie Dedecker. Flexible object encapsulation for ambient-oriented programming. In *Proceedings of the Dynamic Language Symposium - OOPSLA '05: Companion of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. San Diego, U.S.A. ACM Press.* ACM Press, 2005.

[DV04] Jessie Dedecker and Werner Van Belle. Actors for mobile ad-hoc networks. In L.T. Yang, M. Guo, G.R. Gao, and N.K. Jha, editors, *Embedded and Ubiquitous Computing*, volume 3207 of *Lecture Notes in Computer Science*, pages 482–494. Springer, 2004. Embedded and Ubiquitous Computing, International Conference EUC 2004, Aizu-Wakamatsu City, Japan, August 25-27, 2004.

[DVM+05] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-Oriented Programming. In *OOPSLA '05: Companion of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. San Diego, U.S.A. ACM Press.* ACM Press, 2005.

[DVM+06] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-Oriented Programming in AmbientTalk. In *ECOOP 2006: European Conference on Object-Oriented Programming*, Nantes, France, July 2006.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[FMDE04] Tore Fjellheim, Stephen Milliner, Marlon Dumas, and Kim Elms. The 3dma middleware for mobile applications. In L.T. Yang, M. Guo, G.R. Gao, and N.K. Jha, editors, *Embedded and Ubiquitous Computing*, volume 3207 of *Lecture Notes in Computer Science*, pages 312–323. Springer, 2004. Embedded and Ubiquitous Computing, International Conference EUC 2004, Aizu-Wakamatsu City, Japan, August 25-27, 2004.

[FRBAM05] Adrian Friday, Manuel Roman, Christian Becker, and Jalal Al-Muhtadi. Guidelines and open issues in systems support for ubicomp: reflections on ubisys 2003 and 2004. *Personal Ubiquitous Computing*, 10(1):1–3, 2005.

[Gab91] Richard P. Gabriel. LISP: Good news, bad news, how to win big. *AI Expert*, 6(6):30–39, June 1991.

[Gab00] Richard P. Gabriel. Worse is better paper series, January 2000.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1994.

[GJL87] David Gelernter, Suresch Jagannathan, and Thomas London. Environments as first-class objects. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 98–110. ACM, ACM, January 1987.

[GLS98] Jr. Guy L. Steele. Growing a language. In *OOPSLA '98 Addendum: Addendum to the 1998 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, New York, NY, USA, 1998. ACM Press.

[GLvB+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[Gon02] Li Gong. JXTA for J2ME extending the reach of wireless with JXTA technology. Technical report, SUN Microsystems, http://www.jxta.org/project/www/docs/JXTA4J2ME.pdf, 2002.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[Gro02] JSR Expert Group. Java(tm) apis for bluetooth specification 1.0 final release. Java Specification Request (JSR) 82, March 2002.

[GS97] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[Hal85] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[HCC99] Mads Haahr, Raymond Cunningham, and Vinny Cahill. Supporting CORBA applications in a mobile environment. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom-99)*, pages 36–47, N.Y., August 15–20 1999. ACM Press.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, June 1977.

[Hoa73] C. A. R. Hoare. Hints on programming language design. Technical report, Stanford University, Stanford, CA, USA, 1973.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Ithaca, NY, USA, 1994.

[Hui04] Ben Hui. Go wild wirelessly with bluetooth. *Java Developer Journal*, 9(2):26–??, 2004.

[IST03] ISTAG. Ambient intelligence: from vision to reality, September 2003. Draft report.

[JdLT+95] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: a toolkit for mobile information access. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 156–171, New York, NY, USA, 1995. ACM Press.

[Jef85] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[JTK97] A. Joseph, J. Tauber, and F. Kaashoek. Mobile computing with the rover toolkit. *IEEE Transactions on Computers*, 46(3):337–352, March 1997.

[KB92] Murat Karaorman and John Bruno. A concurrency mechanism for sequential eiffel. In *Proceedings of the eighth international conference on Technology of object oriented languages and systems*, pages 63–77. Prentice-Hall, Inc., 1992.

[KB02] Alan Kaminsky and Hans-Peter Bischof. Many-to-many invocation: A new object oriented paradigm for ad hoc collaborative systems. *17th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 2002)*, 2002.

[KCM+01] M. Killijian, R. Cunningham, R. Meier, L. Mazare, and V. Cahill. Towards group communication for mobile participants. In *Proceedings of the 1st ACM Workshop on Principles of Mobile Computing (POMC 2001)*, pages 75–82, 2001.

[KLM+05] Gerd Kortuem, Matthias Lampe, Pedro Jose Marron, Martin Strohbach, and Tsutomu Terada. Workshop on smart object systems, September 2005.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 214–223. ACM Press, 1986.

[Lis92] B. Liskov. Distributed programming in argus. In Akkihebbal L. Ananda and Balasubramaniam Srinivasan, editors, *Distributed Computing Systems: Concepts and Structures*, pages 370–382. IEEE Computer Society Press, Los Alamos, CA, 1992.

[Löh92] Klaus-Peter Löhr. Concurrency annotations. *ACM SIGPLAN Notices*, 27(10):327–340, October 1992.

[LS88] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267. ACM Press, 1988.

[LSC+05] Cristina Lopes, Steffen Schaefer, Siobhan Clarke, Tzilla Elrad, and Jens Jahnke. Workshop on building software for pervasive computing, October 2005.

[Luc87] Steven E. Lucco. Parallel programming in a virtual object space. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 26–34, New York, NY, USA, 1987. ACM Press.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155. ACM Press, 1987.

[MC02] René Meier and Vinny Cahill. Steam: Event-based middleware for wireless ad hoc network. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 639–644, Washington, DC, USA, 2002. IEEE Computer Society.

[McA95] J. McAffer. Meta level programming with CodA. In W. Olthoff, editor, *Proceedings of ECOOP'95, Aarhus, Denmark*, Lecture Notes in Computer Science 952, pages 190–214. Springer-Verlag, Berlin, 1995.

[MCCH05] Holger Mügge, Pascal Cherrier, Pascal Costanza, and Robert Hirschfeld. Workshop on object technology for ami, July 2005.

[MCE02] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. Mobile computing middleware. In *Advanced lectures on networking*, volume 2497, pages 20–58. Springer-Verlag New York, Inc., 2002.

[MCMT06] Holger Mügge, Pascal Cherrier, Wolfgang De Meuter, and Eric Tanter. Workshop on object technology for ami, July 2006.

[Mey93] Bertrand Meyer. Systematic concurrent object-oriented programming. *Commun. ACM*, 36(9):56–80, 1993.

[Mil04] Mark Miller. The E programming language, the secure distributed pure-object platform and p2p scripting language for writing capability-based smart contracts. 2004. http://www.erights.org.

[MMC95] Philippe Mulet, Jacques Malenfant, and Pierre Cointe. Towards a methodology for explicit composition of metaobjects. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 316–330, New York, NY, USA, 1995. ACM Press.

[MMY96] H. Masuhara, S. Matsuoka, and A. Yonezawa. Implementing parallel language constructs using a reflective objectoriented language. In *Proceedings of Reflection Symposium'96*, pages 79–91, April 1996.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[MPR01] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 524. IEEE Computer Society, 2001.

[MT91] Ian Mason and Carolyn Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–328, July 1991.

[MTS05] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, 2005.

[MY90] S. Matsuoka and A. Yonezawa. Metalevel solution to inheritance anomaly in concurrent object-oriented languages, 1990.

[MY93] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. pages 107–150, 1993.

[MZ04] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications with tota middleware. In *Embedded and Ubiquitous Computing*, pages 263–??? IEEE, 2004. Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04), Orlando (FL), U.S.A., March, 2004.

[PST+97a] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'16)*, pages 288–301, Saint-Malo, France, October 1997.

[PST+97b] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 288–301, New York, NY, USA, 1997. ACM Press.

[QR96] Christian Queinnec and David De Roure. Sharing code through first-class environments. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 251–261, Philadelphia, Pennsylvania, may 1996.

[RB99] P. Reynolds and R. Brangeon. DOLMEN - service machine development for an open long-term mobile and fixed network environment, February 19 1999.

[Sat96] M. Satyanarayanan. Fundamental challenges in mobile computing. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 1–7, New York, NY, USA, 1996. ACM Press.

[SBBK95] A. Schill, B. Bellmann, W. Bohmak, and S. Kummel. System support for mobile distributed applications. In *SDNE '95: Proceedings of the 2nd International Workshop on Services in Distributed and Networked Environments*, page 124. IEEE Computer Society, 1995.

[SKK+90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput.*, 39(4):447–459, 1990.

[SM95] Patrick Steyaert and Wolfgang De Meuter. A marriage of class- and object-based inheritance without unwanted children. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 127–144, London, UK, 1995. Springer-Verlag.

[Smi82] Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology, January 1982.

[Smi84] Brian Cantwell Smith. Reflection and semantics in lisp. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 23–35, New York, NY, USA, 1984. ACM Press.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, New York, NY, USA, 1986. ACM Press.

[Tai93] A. Taivalsaari. *A Critical View of Inheritance and Reusability in Object-Oriented Programming*. PhD thesis, University of Jyväskylä, Finland, 1993.

[TK02] Robert Tolksdorf and Kai Knubben. Programming distributed systems with the delegation-based object-oriented language dself. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 927–931, New York, NY, USA, 2002. ACM Press.

[TMY94] K. Taura, S. Matsuoka, and A. Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation. In G. E. Blelloch, K. Mani Chandy, and S. Jagannathan, editors, *Proceedings of DIMACS '94 Workshop*, volume 18. Specification of Parallel Algorithms of *Series in Discrete Mathematics and Theoretical Computer Science*, pages 275–291. American Mathematical Society, 1994.

[TPST98] D. B. Terry, K. Petersen, M. J. Spreitzer, and M. M. Theimer. The case for non-transparent replication: Examples from Bayou. *IEEE Data Engineering Bulletin*, 21(4):12–20, dec 1998.

[TS89] C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. *ACM SIGPLAN Notices*, 24(10):103–112, October 1989.

[UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp Symb. Comput.*, 4(3):223–242, 1991.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.

[VA98] Carlos A. Varela and Gul A. Agha. What after java? from objects to actors. In *WWW7: Proceedings of the seventh international conference on World Wide Web 7*, pages 573–577. Elsevier Science Publishers B. V., 1998.

[VA01] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.

[VDMD05] Tom Van Cutsem, Jessie Dedecker, Stijn Mostinckx, and Wolfgang De Meuter. Abstractions for context-aware object references. 2005.

[VRB99] Werner Vogels, Robbert Van Renesse, and Ken Birman. Six misconceptions about reliable distributed computing. In *HPDC '99: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 36, Washington, DC, USA, 1999. IEEE Computer Society.

[Wal01] Jim Waldo. Constructing ad hoc networks. In *NCA '01: Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'01)*, page 9, Washington, DC, USA, 2001. IEEE Computer Society.

[Wei91] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, 1991.

[WWWK96] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In Jan Vitek and Christian F. Tschudin, editors, *Mobile Object Systems*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 1996.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming abcl/1. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268. ACM Press, 1986.

[ZCME02] Stefanos Zachariadis, Licia Capra, Cecilia Mascolo, and Wolfgang Emmerich. XMIDDLE: information sharing middleware for a mobile environment. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 712–712, New York, May 19–25 2002. ACM Press.