Vrije Universiteit Brussel
Faculty of Sciences
Department of Computer Science
and Applied Computer Science

A Prototype-based Approach to Distributed Applications

Dissertation submitted in partial fulfillment of the requirements for the degree of Licentiaat in de Informatica

Tom Van Cutsem
Stijn Mostinckx

Promotor:
Prof. Dr. Theo D'Hondt

Advisors:
Wolfgang De Meuter
Jessie Dedecker

June 2004

Abstract

With the advent of ever cheaper, more stable and more powerful mobile devices and their integration via wireless networks, a new branch of applications is emerging. The construction of such mobile, flexible and distributed applications place an enormous burden of complexity on the programmer's shoulders. To keep such software maintainable and robust, the programming language in which it is written should introduce the necessary abstractions to master this complexity. Most of the contemporary programming languages fail in providing the necessary language constructs. These languages fall short in their task of making it feasible to create such extensible applications that run in an open, dynamically determined distributed topology.

This dissertation proposes an object-oriented distributed programming language which allows objects to be structured in a flexible and distributed way. The language is prototype-based yielding a more flexible programming medium than its class-based alternatives. The proposed language, called dPico, allows for software to be structured by applying the principle of inheritance in prototype-based languages between distributed objects. Such object relations help us to restrict a number of problems arising in concurrent and distributed environments. The dissertation considers the domains of prototype-based, concurrent and distributed languages. Subsequently, the proposed language is founded with a concurrency and a distribution model. The models selectively inherit concepts from prior programming languages and paradigms.

Acknowledgements

First of all, we would like to thank Prof. Dr. Theo D'Hondt for promoting our research. Special thanks go to our advisors Wolfgang De Meuter and Jessie Dedecker for the many ideas they have proposed, the many discussions that have led to the improvement of this text and for proof-reading this lengthy dissertation. Without them, this dissertation would not be what it is today. We would also like to thank them for having given us the necessary space where we could spend our time discussing and writing this text.

Thanks also to all members of the Programming Technology Lab for the discussions and feedback during the various presentations. Our gratitude also goes to the Department of Computer Science of the *Vrije Universiteit Brussel* for the education and for exposing us to the many facets of computer science.

We would also like to thank our girlfriends Natalie and Sara for their love and support throughout the year. Many thanks go to our parents for providing us with an excellent education and environment to work in. Finally, we would like to thank one another for the good understanding which has made it possible to achieve more together than we could have achieved by ourselves.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advent of increasingly smaller and mobile devices such as cellular phones and PDAs, a growing demand for more flexible software is emerging. Contemporary devices tend to become increasingly more capable of running full-fledged software applications. They have become cheaper, more stable and lots of these devices are not isolated but rather collaborate in increasingly growing networks. The networks themselves have also evolved. Especially with the breakthrough of wireless networks, collaboration between mobile devices has become an affordable reality. Because of the dynamicity involved, a new software engineering domain is bound to emerge if one wants to stay in control of the *software* that runs on these devices and that regulates their collaborations. This will involve the development of new design notations, programming languages, runtime support and so on.

One vision that incorporates software for such evolved computational environments is that of *Ambient Intelligence* (AmI) (ISTAG, 2003). This vision observes an evolution towards personalized small devices that interact with computers that have become invisible and embedded in the environment. This part of the vision is also called ubiquitous computing (Weiser, 1991). Such computing environments surround the user like a "processor cloud", termed a Personal Area Network (PAN).

In order to create such a digital habitat, many hardware-related problems remain yet to be solved. This is more commonly known as the branch of *mobile computing*. This dissertation will focus on the software aspect of Ambient Intelligence. Our main interests encompass the *programming languages* that will be used to develop programs that work in – and interact with – a dynamic, flexible, mobile and open computing world. It is one of the basic hypotheses of this work that contemporary programming languages are not sufficient to cope with the dynamicity engendered by the hardware constellation described above.

1.1 Designing a Distributed Programming Language

The main contribution of this dissertation is the construction of a programming language – called dPico – that is specifically designed to deal with *some* of the

problems imposed by the vision of AmI. dPico is an object-oriented programming language. Yet, it does not feature classes. Rather, it is a classless, so-called *prototype-based* language, where objects are used both to build abstractions as well as to execute programs. dPico is also a *distributed* programming language, where objects running on *multiple* virtual machines can communicate with one another. The language is an offspring of an existing prototype-based language – Pic% – which itself builds upon the programming language Pico (D'Hondt, 1996).

From the description of the vision of Ambient Intelligence, it should be clear that the computing environment is both highly *concurrent* and highly *distributed*, due to multiple programs running on multiple devices. When considering class-based languages in a distributed programming environment, a lot of technical issues arise which render the usability of classes in our envisioned context questionable. We argue in favour of prototype-based languages because they appear to suffer less from the increased complexity of flexible, distributed environments than their class-based counterparts. Our reasons for abandoning classes and resolutely favouring the prototype-based language paradigm will be extensively defended.

When multiple people are running applications on different devices, and these applications are communicating with each other, concurrency naturally occurs. Therefore, the programming language used to build these applications should be equipped with a solid *concurrency model* that brings some order in the chaos of objects concurrently sending messages to each other. This is necessary as concurrent programs have to deal with problems that simply do not exist in sequential languages. Typical concurrency issues are *race conditions* and the *synchronization* of multiple programs. We have devoted a considerable amount of effort to studying existing concurrent programming languages. Since our choice was to use a prototype-based language, we have particularly studied the domain of concurrent prototype-based languages. This domain appeared to be left largely unexplored, with the exclusion of a few notable exceptions, such as the language ABCL (Yonezawa et al., 1986). This language proved to be an important source of inspiration for our own concurrent extension of Pic%, baptized cPico. It has never been cPico's goal to support concurrency to allow for writing efficient parallel algorithms. Rather, it employs a concurrency model that was explicitly designed to model collaborating processes.

The development of dPico has thus been an evolutionary process, going from Pic% to cPico first, trying to conquer the concurrency issues. The additional distribution issues that had to be tackled were subsequently introduced, leading to the language dPico. To be able to know precisely what *kind* of problems one encounters in a distributed computing environment, the field of distributed programming languages was also subject to inspection. We have analyzed several existing distributed languages to see how they cope with the problems at hand.

One of the original contributions of dPico is that it exploits *distributed inheritance* – the structural relationship of *inheritance* or *delegation* among objects applied in a distributed context. As will be argued, the idea of having a "parent" object that can be located on a different machine proves to be beneficial in a dis-

tributed context, especially when this parent object can be *shared* between multiple *children*. Such sharing mechanism is known as *parent sharing* (De Meuter et al., 2003b), and is one of the distinguishing features of many prototype-based languages. This type of sharing seems to offer innovative ways to tackle some of the concurrency and distribution issues by encapsulating *shared state* in a shared parent.

This dissertation will describe the extension of a minimal prototype-based language to a small distributed language. This language was specifically designed to allow for expressive parent sharing between concurrent as well as a distributed objects. In concrete, a dPico program consist of a set of plain objects that may be scattered across multiple virtual machines. Shared state is encapsulated in objects that are modelled as *shared parents*. dPico will provide the necessary object structuring mechanisms to facilitate such organizations.

Notice that our proposed language will not be able to cope with all of the problems that arise in an Ambient context. It has never been our goal to create a full-fledged programming language, since this is clearly out of the scope of this dissertation. Issues such as mobility, distributed exception handling and partial failures are not touched upon. Rather, the main driving force behind this research is the exploration of how well prototypes and parent sharing structures perform in a distributed setting. The overwhelming complexity of an Ambient environment will only be tackled when research from many hardware- and software-oriented fields can be combined consistently. Our contribution is a minimal yet executable and expressive programming language that can serve as an initial model for further research.

1.2 Prologue: An Ambient Intelligence Scenario

We will explore the vision of Ambient Intelligence in a bit more detail by describing a scenario that builds upon one of the scenarios stipulated in the visionary document on Ambient Intelligence developed by the IST Advisory Group (ISTAG, 2003). From this scenario, we will subsequently distill some of the features we deem necessary for a language capable of expressing programs in an AmI context in section 1.3.

In the "Maria - Road Warrior" scenario we follow the interaction of Maria with her P-Com (a hand-held device used to interact with other devices in an AmI-enabled world). The ISTAG scenario sketches Maria's path to a sales pitch she is giving abroad. We see her preparing the meeting, and take the plane to get there, but the meeting itself is largely skipped in the original scenario. This scenario fills that gap and will give the reader a feeling of the type of software applications inhabiting an Ambient world.

Maria enters the seminar room and her P-Com automatically detects and connects to the projector, requesting and downloading a control object. The P-Com adjusts the colour scheme just a bit to match Maria's preferences. The control object is added to her presentation software after which the output of the presentation window is redirected to the projector screen. Maria's other programs remain local on her P-Com.

She opens her sales presentation and activates it. The document asks for her password and begins decrypting itself, as the negotiators of AmbientSoft enter the room. A few minutes later Maria closes the door, signalling that she wants to start. The attending secretary activates her own P-Com and fires up the laser keyboard, ready to take notes.

Maria starts the presentation by introducing her product – a Human Resource Management software package – to the negotiators. Maria knows it won't be easy to convince her audience because of the many competitors on the market. Suddenly, she is disrupted by the secretary's P-Com, suddenly beeping. Apparently a phone call had come in that her software answer machine deemed urgent enough to break the meeting communication barriers. The secretary beams her document along with the running word processor to the negotiator on her left and rushes out to answer the call. The man in turn activates his laser keyboard and signals that Maria can continue.

As the meeting progresses, the situation seemed to deteriorate. Maria feels that the interactive demo programs that were incorporated in her presentation to compare her own package to the competitors' did not have the impact she had hoped for. She figured that her company would not even get the chance for a full presentation for the board of AmbientSoft if she could not convince the negotiators.

She decides to take some risk and to go beyond the interactive demo in an active document. She downloads the prototype of AmbientSoft's customized front-end for her company software. The front-end – designed to run on a workstation – proves to be too heavy for her P-Com. Fortunately, her P-Com detects that the hotel offers a secure deployment server. For safety, Maria adds her own encryption to the program, then migrates it to the hotel server.

Having shown the customized prototype, Maria feels more confident. She knows her product, and is able to highlight its strengths with the full-blown version. Her P-Com had connected the real application with the back-end running on the hotel server and the front-end using the projector control object. This demonstration allows Maria to refute most of the criticisms she receives on her presentation demos.

When the presentation has come to an end, the negotiators start an anonymous vote to evaluate Maria's product. A voting agent is broadcast to every negotiator's P-Com. After all votes have been placed and the results collected, the sales meeting is finished. Tired yet satisfied, Maria leaves the room. Her P-Com goes out of the projector's range, automatically relinquishing control over the projector.

1.3 Required Language Characteristics

Looking back on the scenario, chances are high that the language in which software for AmI will be written will probably not be contemporary languages such as Java or C++. Although it is known from Church's Thesis that all Turing-complete programming languages are equally powerful, we are concerned with the *expressivity* with which a certain problem can be solved in a language. AmI Software *can* be written in contemporary languages but we argue that this will be an extremely difficult task. The complexity that is needed to program the scenario's applications is enormous due to the dynamism that is inherently present in the environment with which the devices interact. More expressive languages are required to better fit these requirements and to make the construction of complex interacting applications easier.

Some language features will now be highlighted that can be observed in the scenario proposed above. A number of these features were already identified in (De Meuter et al., 2003a).

Distributed Object Inheritance Traditionally, inheritance has been used in programming languages to support code reuse. Inheritance in prototype-based languages can be used to support more interesting behaviour, however. The inheritance relation could e.g. be used to "connect" distributed objects. This is illustrated in the scenario via the control object for the projector. This object stays logically connected to the projector. It offers Maria an interface to the projector. Such interfaces can be modelled as views or extensions of the object that represents the actual projector. The parent link of such interface objects can refer to remote objects. Following De Meuter (2004), we believe such parent sharing to be a key ingredient of distributed object systems. The key difference between parent sharing and sharing through composition is the sharing relation itself: the composition link is replaced by a *delegation* link, allowing for safer, more encapsulated and bidirectional communication. In (Dedecker and De Meuter, 2003) these ideas are further explored in the context of mobile agents.

Strong Code Mobility The scenario also introduces *strong code mobility* which implies that running programs can be *migrated* or moved from one machine to another. When the secretary beams her running word processor to another person, the word processor itself continues to work as if nothing happened. This kind of migration is not achieved with for example today's applet technology, since applets need to be restarted when they arrive at a new machine. In such models, the programmer has to write his software in partitioned "states" depending on the location where the code is executing. In an environment where objects and code can be moved so freely, this would quickly lead to code which is hard to understand, develop and maintain. Although Strong Mobility is an important language feature, we will not consider it in detail in this dissertation.

Broadcast Communication The voting agent is concurrently broadcasted to the P-Com of all negotiators. This sort of communication could be achieved expressively by being able to send a message to some *"collection"* of objects, representing the identity of all registered P-Coms. A reception system is needed to be able to submit all votes safely upon completion. Multivalues (De Meuter et al., 2003a) provide for a suitable representation of such dynamic collections of objects. Although we have performed several gedankenexperiments regarding multivalues, this dissertation will not consider them any further.

Autonomous Processes A language for AmI environments should cope with *concurrency*: multiple programs will be running multiple tasks simultaneously. Such languages need constructs to easily create and manipulate processes. Ideally, these processes should be as autonomous as possible, being able to migrate quickly to other Ambient devices. Communication between processes should be made transparent (with respect to e.g. location of execution) to conquer the complexity inherent in an Ambient world. Much of our work has revolved around the creation of such a process model.

As already explained, not all of these language features have been investigated in our work, but listing them explicitly *does* give the reader a good feeling about the type of language we envision. Our work has to be considered as a small, yet essential, constituent of such a language. It investigates the interaction between prototypes, delegation, concurrency and distribution.

1.4 Language Engineering Approach

The design of a concurrent and distributed programming language has lead us to consider both language *feature* design, which is concerned with *innovation*, and *language* design, which is about *integration*. The difference between both is stressed in the classical paper by Hoare (1973). We wish to uncover some of the key features that allow programmers to write elegant, readable, maintainable and robust programs that can cope with a dynamic environment. Subsequently, it should be ensured that the implemented features integrate well into an existing prototype-based language. We have tried to combine both the activities of innovation and integration.

Striving for *minimality* in the language is important if one does not want to end up with language features interacting in non-predictable ways. This vision towards language engineering is defended in (Smith and Ungar, 1995), where the authors extensively report on their experiences in designing the programming language Self. Smith and Ungar (1995) coin the term *the language designer's trap* to denote the temptation of language designers to add features because they merely allow for appealing examples. Concepts should be added to a language sparingly, if the language is to remain clear for the programmer. Moreover, "example programs" that cannot be covered by existing language features should yield in a redesign of

those features, not in a reflex to add new features to cover them.

Throughout the dissertation, we have taken an *iterative* and *experimental* approach to language engineering, where the final language is the product of a sequence of iterations. Each iteration is supported by a prototype implementation. Such an iterative approach, augmented with experiments conducted in the prototype implementation, allows for inconsistencies to be detected earlier such that language features can be refactored in subsequent iterations. A consequence of this approach is the gradual extension of Pic% in two parts. First, the design and implementation of cPico – a concurrency model for Pic% – is discussed. Only then a distribution model is built on top of the constructed concurrency model.

1.5 Dissertation Roadmap

The dissertation starts out with an extensive overview of the fields of prototype-based, concurrent and distributed programming languages. Having established the necessary background, we will turn to our own research results, featuring the extension of the prototype-based language Pic% to a concurrent respectively distributed programming language.

The field of prototype-based object-oriented programming is introduced in chapter 2. Chapter 3 will then introduce some of the problems that arise when writing concurrent object-oriented programs. Subsequently, chapter 4 will introduce relevant distributed programming languages. We will find that these languages introduce problems of their own, leading to new issues in language engineering. We will discuss where classes fail to solve these issues, while prototype-based languages are able to cope with some of them in section 4.4.

The creation of dPico is spread out over two chapters. First, the concurrency model incarnated in cPico is discussed in chapter 5. The distribution aspects are subsequently added in chapter 6. The result is the distributed programming language dPico, incorporating distributed object inheritance through parent sharing. The language's design (and to a lesser extent, its implementation) are discussed, as well as its limitations.

The dissertation is concluded in chapter 7, reflecting on the achievements and proposing future work together with some perspectives for future research. Interested readers will find an informal specification of the Pic% language and our extensions in appendix A. These semantics form the basis of our implementation of an interpreter, written in Java. Finally, appendices B and C conclude the text with a couple of programming examples in cPico and dPico and an overview of both languages' most important concepts.

Chapter 2

Prototype-based Languages

2.1 Introduction

In the previous chapter we have given the reader a basic idea of the problems we wish to solve by means of a scenario. As we have already mentioned we will use prototype-based languages to reach this goal. We believe delegation and sharing to be key features of prototype-based languages. This chapter will introduce prototype-based languages thoroughly but with respect to the way we have used them to support our thesis.

In section 2.2 we will first introduce how the world is modelled from a prototype-based perspective. We will then continue by comparing class-based and prototype-based languages in section 2.3. This will introduce a more thorough discussion of what constitutes a prototype-based language. We will devote section 2.4 to that purpose, which concludes the abstract-theoretical introduction of prototype-based languages. We will then provide an overview of the history of prototype-based languages in section 2.5. In this overview we will have special attention for three cases, which will be elaborated. Section 2.6 concludes our discussion on prototype-based languages.

2.2 Modelling Our World

In the current-day modelling of applications, objects take a prominent place. More and more programmers are starting to use object-oriented programming languages, mostly due to the success of languages such as C++ (Stroustrup, 1986) and Java (Gosling et al., 1996). It is also due to the fact that those languages are so popular that the impression may exist that object-oriented programming should automatically lead to the use of class-based languages such as the aforementioned Java or C++ languages. This is far from true, when examining the space of object-oriented languages with disregard of popularity, we can see that class-based languages form only half of the spectrum of object-oriented languages (De Meuter, 2004).

This means that there is another, far more unknown sibling in the world of

object-oriented programming. This chapter will introduce the basic concepts of this sibling, which is called *prototype-based programming*. Programming and more specifically designing applications with prototypes requires essentially different techniques than the design techniques used in class-oriented design. This is why we will first explain how the prototype-based view of the world differs from the more traditional class-based view of the world. We will use a review in the PhD dissertation of De Meuter (2004) as a starting point. This review in turn is based on the outstanding article by Taivalsaari (1996).

The fact that the world has been – and still is – dominated by the class-based view of the world is not surprising. It is in fact a reflection of a way of viewing the world which has prevailed for over 2000 years. This view, which is the Aristotelian view of the world, stems from the ancient Greek philosophers Plato and Aristotle. They categorized the world into objects which were in fact instances of ideas. These ideas were abstract entities, which perfectly captured what is essential to be for instance an elephant. And then any elephant was an instance of this "genus" and since there are no ideal instances there was also some variation which was denoted by the term "differentia" (corresponding to our current notion of *inheritance*). We can easily observe that this way to *classify* the world is very closely related with the class-based way to organize software. We try to describe by means of a class, what we consider to be the abstract "genus" of the objects we wish to describe.

However, some criticism to this approach seems justifiable. In (Taivalsaari, 1996) a lot of problems are presented concerning the view of the Greek school. The interested reader will find a good overview in this article. We will illustrate what is relevant in this context by a small experiment. We ask the reader to think of an elephant, before reading on. Chances are high that most readers will not have spontaneously thought about an abstract classifying description, such as "a large grey mammal with a trunk". Most people think about most concepts, in particular the ones which are not mathematically formalized, in terms of a few "representative" instances. This theory of prototypes was amongst others defended by the twentieth century philosopher Wittgenstein. This approach, based on resemblance to a specific *prototype*, found its way to computer science initially through the development of frame-based languages for knowledge representation in AI applications (mainly developed by Minsky).

An influential paper by Lieberman (1986) introduced this work into the class-dominated research area of object-oriented programming. From that point on, the language theoretical perspective took over in the history of prototype-based object-oriented languages, in a continuing search for smaller, simpler and conceptually cleaner languages (De Meuter, 2004). This search will be illustrated by the historical overview in section 2.5. Before we will present this overview we will provide a more language-theoretical introduction to prototype-based languages. The next section will do so by illustrating the differences with class-based languages. We will continue then in section 2.4 by discussing the features that a prototype-based language provides in order to allow the programmer to model the world.

2.3 Classes Versus Prototypes

To explain the basic properties of a platform that features prototypes instead of classes, we deem it useful to allow the reader to discover the differences in the paradigm by initially holding both paradigms next to each other. We will explore the differences using the concepts that were introduced in the Treaty of Orlando (Lieberman et al., 1987), a taxonomy paper that was written on the occasion of the OOPSLA conference in Orlando, Atlanta. A similar summary of this treaty can also be found in (De Meuter, 2004).

Two basic concepts were introduced by the Treaty: *Empathy* and *Templates*. Empathy is used to describe how we express that an object "resembles" another object, and thus reuses behaviour of this "parent entity". *Templates* generate objects from their own image. This concept will thus allow us to generate several objects which are all very much alike each other, possibly with the exception of a small amount of state.

The Treaty then describes several ways these concepts can be filled in. Empathy is implemented by prototype- and class-based languages using respectively the *delegation* and the *inheritance* relation. We assume that the reader is familiar with the concept of class inheritance.

Delegation was first introduced by Lieberman (1986), to describe the typical behaviour of an object in a prototype-based system which does not understand a message. This object will then delegate the message to its parent, while passing a reference to itself along, such that the receiver of the message does not change. It is as if the receiving object implicitly states: "I don't know how to handle [this] message. I'd like you answer it for me, if you can, but if you have any further questions, [...] or need anything done, you should come back to me and ask" (Lieberman, 1986). The object is passed along with the request, and due to the "late binding of self", it will serve as a callback for future `self` sends. This "late binding of self" is what makes delegation distinct from ordinary message forwarding[1] as is shown in figure 2.1.

Prototype-based languages, like class-based languages introduce a *self* pseudo-variable which refers to the object that has received the message. This allows for the objects to which a message is delegated to query the receiver, should they need further assistance. We can envision the scenario as follows. An object receives a message m, but it has no definitions of it. This means that the receiver itself cannot handle the request. Therefore it delegates the message to some other object usually termed a *proxy*, and it supplies its *self* as some sort of callback, should the parent need additional information, such as instance variables. In our context, this proxy object will *always* be a parent object. Delegating to "parent" objects is also termed *object inheritance*.

The Treaty of Orlando defines several degrees of freedom for this empathy

[1] Unfortunately some confusion exists about delegation since the Delegation Design Pattern (Gamma et al., 1995) actually explains message forwarding.

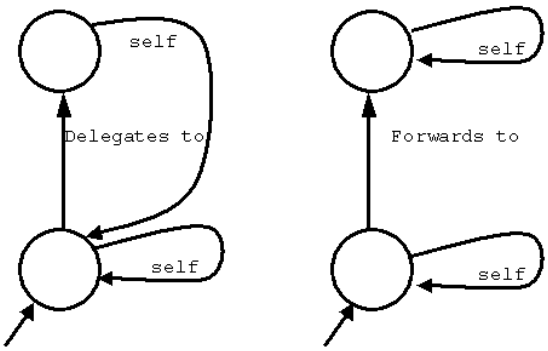

Figure 2.1: Late binding of Self

This figure illustrates the difference between delegation as proposed by Lieberman (1986) and message forwarding also called delegation in (Gamma et al., 1995). When delegating all subsequent self sends will thus be sent back to the original message receiver.

relation, which we shall discuss when we will come back to the matter of delegation and sharing in section 2.4.4.

The second concept that was introduced in the Treaty is the concept of a *template*. We define as templates any structure that allows us to create objects with a given desired structure. In class-based languages this is achieved through class instantiation so there we use classes as templates. In prototype-based languages we have abandoned the notion of classes, in favour of the simplicity we get when we only have to deal with objects. Thus if we see a particular *prototype* which we want to use as a template for a new object, this can be achieved by *cloning* that particular object.

Finally Lieberman et al. (1987) also make a distinction between static and dynamic templates. This means whether an objects structure can be modified after creation. In class-based languages a template is typically static. We will discuss the consequences of taking either option for the case of prototype-base languages in section 2.4.3. Table 2.1 summarizes the differences observed between class-based and prototype-based languages.

	Class-based languages	**Prototype-based languages**
Empathy	through inheritance	through delegation
Templates	by instantiating classes	by cloning objects

Table 2.1: General comparison between class- and prototype-based languages

2.3.1 Prototype-based Languages: Advantages

We have introduced the general flavour of prototype-based programming, by introducing the two core concepts, delegation and cloning. We will now present the advantages that prototypes introduce through the use of these simple primitives. Similar enumerations of the advantages of using prototypes are presented in (Borning (1986), Ungar and Smith (1987), De Meuter (2004)). The most general observation we can make is that programming with prototypes often simplifies the problem.

Prototypes simplify the concepts that are present in the design of an application, since we should only consider objects, and the only relations that exist are the *delegates-to* relation and regular composition.

Prototypes allow per-object specification of behaviour. This is useful for expressing a singleton pattern, as unlike in class-based languages we do not need an extra class, which is often useless apart from creating the one instance. Furthermore, we can also attribute different behaviour to one object, such as debug code. In a class-based language this code for a specific faulty object would also affect all other instances, which is often a huge disadvantage.

Due to cloning, we are able to avoid the case where we have fields in our newly generated object that are not filled in. In class-based languages we perform initialization through a constructor call. This call will then specify a plan to create a new object. Prototypes offer a different metaphor, cloning, which automatically ensures that no variable is left uninitialized.

One of the contributions of using prototypes, that is often overlooked is that there is no longer need for an infinite meta-class regress. Most class-based programs do not suffer from this problem but that is because they do not offer full-fledged meta-programming. In SmallTalk (Goldberg and Robson, 1989) we can observe a sophisticated solution to avoid a potentially endless regress of meta-classes. Whereas we consider this solution a pearl in language design, we nevertheless want meta-programming to be *simpler* to understand. The main problem is that in class-based languages, objects are not self-sufficient as they also require their class to be fully specified. Prototype-based programming offers us self-sufficient objects to avoid that when a programmer starts to master the language, he comes across hidden difficulties, such as a meta-class hierarchy. With prototype-based languages, he can work with objects at every level.

2.3.2 Prototype-based Languages: Drawbacks

Prototypes are not the holy grail of programming languages, in fact they also have significant shortcomings in their most primitive form. We will present here a set of “features” that we find in class-based languages, but which seem to be missing in some prototype-based languages. These issues are addressed in the Agora-language family, which will be introduced in section 2.5.2. The language we have used to support the concepts we will introduce in this dissertation, Pic% is a heir

of the Agora family, and will be thoroughly introduced in section 2.5.3.

Classification of objects is a powerful tool which is offered by class-based languages in a very natural way. When writing programs it is often important to be able to speak about a set of related objects, which share either representation or just a common interface. In a simplistic prototype-based model that uses only delegation and cloning it is nearly impossible to keep track of how objects are related, as there is often no relation between an object and the object that it was cloned from.

Another important feature of class-based languages is that they allow us to reason about concepts that are inherently abstract. This can often be impractical, since in most cases representing concrete things in terms of abstract classes requires a usually difficult and time-consuming *analysis* phase[2]. Prototypes however enforce that everything we describe is concrete, i.e. in terms of a concrete realization of some prototype. When it comes to domains which are severely formalized this can be a problem as well. Everybody knows what is a queue or a stack, but can we say that stacks or queues are actually clones of a certain prototype? It seems a bit awkward to reason about some ideal (prototypical) stack.

Finally, classes decouple an object's representation from its behaviour. This makes it possible to more easily protect the behaviour. Though this decoupling can often be problematic (cfr. the problems mentioned in section 2.3.1), we can equally foresee some problems if we declare a whole set of objects depending on some prototype, and this prototype all of a sudden gets modified[3]. We require some sort of protection on our prototypes, though we do not want to forbid all changes to the shared parent, as we consider them to be essential to programming with prototypes.

We have now introduced some of the problems that arise when programming with prototypes. These problems have been dealt with by several prototype-based languages over time. We will present those solutions in section 2.5 which describes a short overview of the history prototype-based languages. To provide a set of concepts to assess these different solutions with, we will describe a taxonomy of language features, that characterize different forms of prototype-based languages.

2.4 Prototype-based Language Features

Over time a wide variety of prototype-based languages has been developed, with different language features. This may lead to the discomforting feeling that we can no longer see the global picture of what prototype-based languages are actually all about. Dony et al. (1992) have therefore tried to develop a taxonomy that allows us to define in a well-formed way exactly where all these languages fit in. We think their article provides an excellent basis to reason about prototype-based

[2]Prototypes lend themselves to a different model of software development where this phase can be largely skipped, namely Rapid Prototyping Development (De Meuter, 2004)

[3]This problem was already observed in (Lieberman, 1986) and was dubbed the *prototype corruption problem* by Blashek (1994).

languages, and we will in this section summarize the core concepts introduced in their taxonomy paper.

2.4.1 Slots and Variables

The first significant distinction we can make when we try to divide prototype-based languages, is in how they handle variables and methods. Two approaches exist, the first – classical – one treats variables and methods differently, just as most classical class-based languages do. This allows for standard mechanisms for visibility to be applicable in these languages to protect the encapsulation of objects.

Another approach unifies the treatment of variables and methods by unifying them into one concept called *slots*. This is the approach that is advocated by languages such as Self (Ungar and Smith, 1987) and Agora (Codenie et al., 1994), which will both be discussed later on respectively in sections 2.5.1 and 2.5.2. When using slots a variable is "expanded" to a couple of accessors methods. This solution is indeed a good one since it promotes simplicity, through both a reduction in the number of available concepts, and by reducing variable accessing to a method invocation. One problem that remains is that when we use slots we have to define a new mechanism to restrict visibility of slots, to ensure that encapsulation of objects remains possible. In the following section we will talk about *fields* to denote respectively slots, or a combination of variables and methods, to abstract from the path taken in this first branching point.

2.4.2 Object Creation

Object creation is another important matter when designing an object-oriented language. Two different approaches exist with respect to how objects are created. The first one states that any object should have a "parent". This means that we have some sort of Root object in the system. We see this phenomenon in class-based languages as well, for example the `Object` classes in Java (Gosling et al., 1996) and Smalltalk (Goldberg and Robson, 1989).

The alternative is to allow ex-nihilo creation of objects. This means that definitions of objects with no parent are allowed. In class-based languages we find that C++ (Stroustrup, 1986) follows this approach. Should we allow ex-nihilo creation, we also need to decide whether we only allow the creation of empty new objects, or whether we allow new objects to have some initial behavior. It may seem strange to consider the creation of empty objects, but this should be seen against the light of the dynamic modification, which we will discuss in the next section.

2.4.3 Dynamic Modification

One of the most important issues to decide when defining the semantics of a prototype-based language, is to decide whether or not we allow dynamic modification of object structure, i.e. adding and removing fields at runtime. This decision

should be considered with great care as its repercussions on the final language are huge. Therefore we will go into more detail here, to explore what solutions exist. First of all, we will explain why this decision is so important.

2.4.3.1 Reasons for Dynamic Modification

When we decide on whether we allow dynamic modification in our language this has repercussions on the aforementioned object creation. This is also why it is actually the second branching point in the hierarchical taxonomy (Dony et al., 1992). If we do not allow dynamic modification we can rule out a set of trivial languages that do not allow ex nihilo creation, or that only allow creation of empty objects. Similarly if we allow dynamic modification we can reduce the case where we have creation with initials to a shorthand for creation of empty objects with subsequent modifications.

Adding dynamic modification to a prototype-based greatly enhances the flexibility that is offered by the language. Since in most cases it is exactly this flexibility that leads us to using prototypes in the first place, this is indeed a powerful feature. Moreover it also has a few very concrete benefits for the practice of software engineering itself. Consider a running program in which we spot some unexpected behaviour. If dynamic modification is allowed in our language we can specify new behaviour at runtime[4] of some objects which gives us additional information about for example the state of the objects etc.

2.4.3.2 Forms of Dynamic Modification

The most basic form of dynamic modification is the addition and deletion of fields, on an object in a running system. More elaborate forms exist for example in Self, which makes the parent slots first class entities. In Self we can thus assign new objects to act as the parent to which we delegate our messages. We elaborate further on this in section 2.5.1.3 to show how this facility allows the expressive translation of the State Design Pattern described by Gamma et al. (1995).

2.4.3.3 Repercussions

Allowing dynamic modification in a prototype-based language offers a lot of power to its users, but it is also one of the main sources to the justifiable criticism that most prototype-based languages (especially those that allow *wild* modifications) are too "flexible" to create realistic software with (De Meuter, 2004). This criticism will be dealt with by the Agora-family of prototype-based languages which will be discussed in detail in section 2.5.2.

[4] We can for example replace a method body by first removing the method and then re-adding a method with behaviour that allows us to debug the object

2.4.4 Delegation and Sharing Mechanisms

Another very important decision is how we implement *sharing relations* in the language. Sharing is one of the key features of prototype-based languages, and thus many variants exist on this theme. In our context sharing will prove to be an essential part of our model as we will demonstrate in chapter 6. Because choosing one specific solution has a considerable influence on the semantics of the language, we will devote a considerable amount of space to this topic. First we will introduce a taxonomy of sharing relations. Afterwards we will look at the ways these sharing relations are achieved in different languages.

2.4.4.1 A Taxonomy of Sharing

We will divide the set of sharing relations based on five criteria, compiled from several sources. The last three rules correspond to the degrees of freedom for empathy, or sharing as we prefer to call it, identified in the Treaty of Orlando (Lieberman et al., 1987). These concerns are presented last since they are only relevant to *life-time sharing* relations, which are introduced below.

What The first criterion in the taxonomy is based the on the three different types of sharing identified by (Bardou, 1996). These types of sharing are distinguished based on *what* is shared.

1. *name sharing* implies that objects share an interface, this means that objects have a field with the same name.
2. *property sharing* means that the objects effectively share the field, this means that the parent object has the field, and the child accesses it through a delegation link or a similar mechanism.
3. *value sharing* means that both objects each (conceptually) have a field, which points to the same value. The difference between both will become clear when we introduce Split Objects in section 2.4.5.

When The next criterion focusses on *how long* the sharing relation holds. In doing so (Dony et al., 1992) can distinguish between two types of sharing

1. *creation-time sharing* corresponds to the *cloning* metaphor in most prototype-based languages. It is a sharing relation where the sharing is only explicit at the creation time of the object. This means that any subsequent change to the object that served as a prototype is not reflected in the new object. As such the only sharing relation of the ones we have just introduced that allows this is *value sharing*.
2. *life-time sharing* . Life-time sharing means that the link between both objects is not disconnected. Typically this type of sharing is achieved by means of *delegation*, though this is not the only way as will be illustrated shortly, when we introduce the non-delegating languages.

Dynamicity This concern expresses whether the sharing relation can be modified at run-time. Self is an example is a language that allows such dynamic modifications of the delegation link of its objects.

Explicitness We wish to know whether the sharing needs to be expressed explicitly or whether it is an implicit feature. We will discuss the different alternatives with respect to this fourth property in the next sections, when we discuss in detail the ways sharing can be achieved in prototype-based languages.

We will use the taxonomy we have just introduced to look at how prototype-based languages can implement these sharing relations.

2.4.4.2 Non-delegating Languages

In our introduction we have explicitly not talked about a "delegation" relation, though this is a sharing relation, as is noted in (Bardou, 1996). Delegation is without a doubt the single most common one for prototype-based languages, but languages such as Kevo, which was developed in the PhD dissertation of Taivalsaari (1993), do not delegate messages but use a *concatenation* strategy. This means that when we specify a new object by extending an object with a set of new slots[5], we actually copy down all the slots of that object.

This implies that when changes are made to the object which was the target of extension, the extended object will not see this. Kevo's concatenation thus introduces creation-time value sharing, not life-time sharing. Kevo in fact inverses the roles attributed to cloning and extension by Dony et al. (1992) since Kevo's cloning operator causes life-time sharing, not through the extra indirection that the taxonomy proposed, but via the use of first-class *cloning families*.

Cloning families allow us to maintain a reference to all objects that are similar to the object in question. This reflect the original ideas by Wittgenstein which proposed a link between objects based on *similarity*. Properties can then be manipulated both on per-object level, and per-group level. By means of the per-group level manipulations we create a form of explicit life-time value sharing.

Obliq (Cardelli, 1994) is another example of a non-delegating language, it uses the term *embedding* instead of concatenation, but the concept is similar. Obliq does not have cloning families, however, as the embedding strategy is employed with distribution in mind. Obliq will be discussed in detail in section 4.5.4 as part of the existing distributed languages presentation.

2.4.4.3 Explicitly Delegating Languages

Next to the concatenation strategy of Kevo and Obliq, there is also the alternative of making delegation explicit. This necessitates a way to intercept the messages

[5]This is by all means equivalent to defining an object that delegates to a parent

that we want to delegate, as well as a way to express delegation without confusing it with ordinary message sends (Dony et al., 1992). The difference between both is important due to the different treatment of `self`. At first it may seem cumbersome to force the programmer to explicitly control which messages should be delegated, but on the other hand it does add to the power of the language since we can specify more difficult delegation schemes by delegating messages to different "parents".

An example of such a language that offers a form of explicit delegation is MOOSTRAP (Mulet and Cointe, 1993). The design goal behind MOOSTRAP is to provide a reflective kernel for a prototype-based language. The language itself does not have delegation, but its reflective capabilities can be used to introduce delegation, through specification of a delegating meta-object. MOOSTRAP uses an approach that is quite unique as it reifies a message send as a two phase process of lookup and application which was first introduced in (Malenfant et al., 1992).

For lookup, a method in the meta-object is used[6]. Thus the user can introduce seemingly implicit delegation by defining a delegating meta-object. However since the lookup is reifyable, the programmer can implement any delegation scheme he wishes, which makes this a form of explicit delegation. We will discuss the sharing potential of delegation after having discussed implicit delegation, since this is basically the one we will mimic using the meta-techniques of MOOSTRAP.

2.4.4.4 Implicitly Delegating Languages

Most prototype-based languages qualify for this category as they implement a variant of the implicit delegation scheme that is introduced in (Lieberman, 1986). This means that messages that are not understood at the current level are automatically delegated to the parent object. We have already introduced this form of delegation in section 2.3, when we have compared it to class-based inheritance.

Delegation automatically introduces a "parent sharing" relation as is discussed in (Bardou, 1996). How we fill in this life-time sharing relation will affect whether we support Split Objects or not. This final step in the taxonomy is the topic of the next section.

2.4.5 Split Objects

We will now turn our attention to the second choice in our taxonomy of sharing, namely *what* is shared. This decision is also represented in the taxonomy, as the choice of whether or not the language handles split objects.

A split object (Bardou and Dony, 1996) is a set of objects connected through parent links. The typical example consists of a person object, that has for example a name and some personal attributes. This person appears in different roles to the outside world, to his employer he is an employee, to his wife, he is a husband, etc... Each of these roles has specific attributes, but however there is but one conceptual entity/identity. In this case we will support this by letting the roles

[6] which shortcuts its own lookup, to avoid infinite meta-regress

share the “properties” of their parent. This means that if changes are made through one of the roles an object plays in the system, these changes are reflected in the parent. Figure 2.2 shows how split objects interact, and shows their relation to the single real identity, which is the person object.

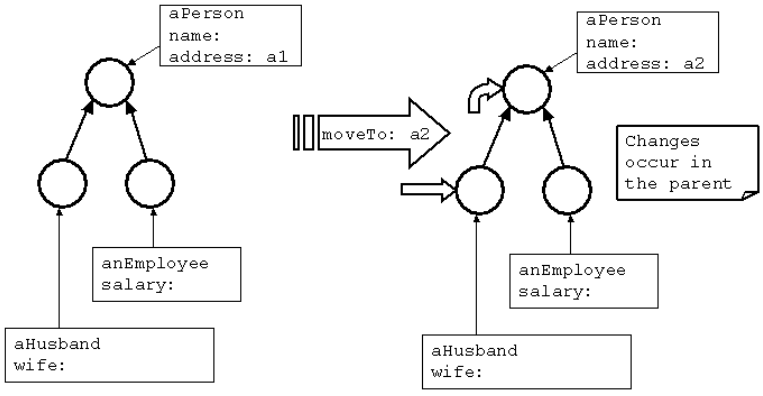

Figure 2.2: Split Objects

Another option is to view the extension by delegation, which is known as the *differential copy* approach (Bardou, 1996). Consider we know an elephant named Clyde and later we learn about another elephant Fred. We then create a hierarchy in which we refer to Clyde for information we have no specific knowledge about concerning Fred. This is illustrated in figure 2.3. Now consider Fred loses a leg, we of course do not want all elephants to suddenly have three legs.[7] In this case we deal with two identities being Clyde and Fred, and we will actually let child objects share the “values” of their parent. This means that conceptually we have a “lazy copy-down” of slots when they are changed in the child. The prototype-based language NewtonScript (Smith, 1995) for example uses such a “value sharing” strategy (Bardou, 1996).

2.4.6 Conclusion

We have used the taxonomy of Dony et al. (1992) to introduce in this section the concepts that need to be considered when designing a prototype-based language. First of all we should decide whether we want to unify variables and methods in slots. Furthermore we should also consider how objects are created, and whether it should be possible to modify the structure of objects, after they have been created. Another very important issue is what type of sharing relation we impose between different objects. We have devoted a considerable amount of space to provide some insight on the different solutions that exist, as this choice is from our point of view

[7] note that this example is the exact inverse of the one in (Lieberman, 1986) which deals with the *prototype corruption problem* (Blashek, 1994)

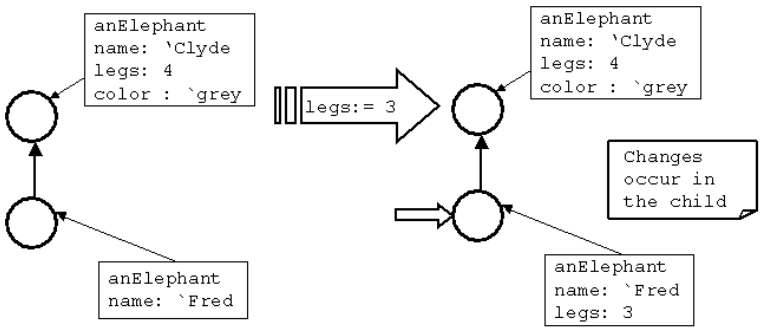

Figure 2.3: Differential Copy Delegation

the most important choice to make. Finally we have also reviewed the support for split objects, which is related to the aforementioned sharing relation chosen. The concepts we have introduced may be a little abstract, which is why we will ground them, by discussing a few prototype-based languages in the next section.

2.5 An Overview of Prototype-based Languages

In this section we will provide an overview of the rich history of prototype-based languages. Throughout this text we have already briefly introduced languages such as Kevo and MOOSTRAP which have guided us to important insights in the diversity of prototype-base languages. Though there are many languages that deserve a treatment of considerable length, such as JavaScript, NewtonScript, Garnet and Exemplars, we will refrain from including them here, due to space restrictions. Other object-based languages of interest, such as Obliq and ABCL will be introduced when we review the state of the art of distributed, respectively concurrent languages.

In section 2.5.1 we will focus on Self, which is one of the most complete and powerful prototype-based languages available. We will continue in section 2.5.2 with a presentation of the Agora language family which introduced several concepts that we consider to be extremely important, and which we will faithfully integrate in the language we will use as a starting point. This language, called Pic% introduces the principles from Agora into the simple yet expressive Pico language. We will introduce this language in section 2.5.3.

2.5.1 Self: The Power of Simplicity

2.5.1.1 Introduction

Self (Ungar and Smith, 1987), mainly developed by Sun Microsystems, is the most evolved prototype-based language to date. It can in many ways be regarded as a

prototypical prototype-based language. Self can be regarded as a transposition of Smalltalk into the prototype-based world. Yet, Self is significantly more minimal and simpler than its class-based counterpart.

Self, mainly designed for exploratory programming strives for *uniformity* in all of its language features. It is this devotion to uniformity that gives rise to Self's great expressive power (Ungar and Smith, 1987), and also one of the reasons this language is considered to be a pearl of programming language design (De Meuter, 2004). Ungar and Smith (1987) believe that reducing the basic concepts in a language can make the language easier to explain, understand and use.

In this section, we will try to give a concise overview of the core language concepts, continuing with how method activation is unified with prototypes. Furthermore, we discuss some interesting programming techniques (or idioms) which first emerged through programming experience in Self. We will show that much of these idioms give rise to more flexible or expressive object structures than can be achieved in their class-based counterparts.

2.5.1.2 Concepts

This section outlines the core concepts of the Self language. Since Self strives for minimalism, it is not surprising that there are only two fundamental entities: objects and messages. Other language features, like the notion of a variable or control structures like "if-then" statements have been repolished to fit the object-centered paradigm smoothly.

Messages subsume variables In Self, computation proceeds by message passing along objects. Message passing in Self is *the* fundamental operation (Ungar and Smith, 1987). The language does not contain variables. Instead, objects are stored in so-called *slots* which can be accessed or modified by message passing. When one declares a slot in Self, the system will automatically generate the appropriate accessor and mutator methods. Slots can be declared constant, in which case no mutator is generated. Variable access is thus entirely replaced by message passing: an object can just send the accessor message of the specific slot to oneself. An object can access itself through the keyword `self`, but this keyword may be omitted when performing self-sends, making this variable replacement scheme as concise as in other languages.

By unifying variable access with message passing, Self eliminates the distinction between both, effectively making message passing and inheritance a more powerful operation (Ungar and Smith, 1987). The reason is that children of a given object can *refine* the accessor methods, thus, being able to provide those "variables" with a totally different behaviour. This simple scheme realizes variable overriding, simply because variables are replaced by methods. In many other object-oriented languages, such as Java, variable overriding is impossible due to the different treatment of variables and methods.

Yet another advantage of the message passing scheme is the treatment of scoping rules. Since a variable access is replaced by an ordinary self-send, all method lookup rules apply to "variable access", which actually boils down to simple nested scoping rules: if a slot is not found in the receiver, it is looked up in the parent. Note that this also implies that objects can see their parents slots.

Objects and slots Three types of objects inhabit the Self world: ordinary objects, method objects and block objects (De Meuter, 2004). Objects can be created ex-nihilo by just listing a number of slots between bars, separated by dots (a so-called *slot list*). A point object can thus be represented by `|x. y|`. If such a construct is followed by an expression, the result is a method object. The declared slots will then act as local "variables" for the method. Local slots can also be preceded with a colon, making the slot a parameter of the method. Self provides proper syntactic sugar to inline these parameters in the method name. This is similar in spirit to Scheme's `define` shorthand for lambda expressions (De Meuter, 2004). An example will clarify the syntax:

```
method: arg = (|x <- 0 | arg dosomethingwith: x)
```

declares a slot named `method` which contains a method object, having a parameter `arg` and a local slot `x`.

Objects can *delegate* messages to parent objects. Parents can be denoted by annotating a slot with an asterisk. This will automatically forward messages not found in the receiver to the parent, but with late binding of self: the `self` pseudovariable will still point to the original receiver. Thus, Self implements delegation as put forward by Lieberman (1986). Self implements *multiple inheritance*, which means that multiple slots can be marked as parents, in which case method lookup is governed by more complex rules to disambiguate slots in multiple parents [8].

As a more complete example, consider the following stack implementation in Self, taken from (Tolksdorf and Knubben, 2001):

```
aStack <- (|
  stack = array clone.
  top = 0.
  push: obj = (top: (top+1). stack at: top Put: obj).
  pop = (top: (top-1)).
  getTop = (stack at: top)
|)
```

Closures A third type of object is the *block object*, which is the equivalent of a so-called *closure* which encapsulates a method together with a pointer to its enclosing

[8] Multiple inheritance has the reputation of being rather complex. In retrospect, Smith and Ungar (1995) admit that multiple inheritance was a feature that had probably better been left out.

scope (Ungar and Smith, 1987). This scope is then used to resolve references to free variables. Such a block object is useful for passing around code together with its environment of definition (the *lexical environment*) (De Meuter, 2004). Blocks are created just like methods, but are surrounded by brackets instead of parentheses.

Blocks are also found in the Smalltalk language, where they are used to define control structures (since blocks allow for *lazy evaluation*, which bypasses the default *eager evaluation* of arguments). Self follows this approach and implements selection ("if") and iteration control structures in terms of messages taking block objects as arguments. One can for example define the selection control structure by implementing the following method in the boolean object `true`:

```
ifTrue:  t ifFalse:  f = (t value)
```

Blocks can be evaluated by sending them the `value` message. This is similar to performing the `apply` operation on a function in e.g. Scheme. To use the described control structures, the caller must pass its arguments wrapped in a block:

```
q empty ifTrue:  [ 0 ] ifFalse:  [ q dequeue ]
```

Self is more powerful than Smalltalk with respect to this use of closures. As in Smalltalk, the user can define his own useful control structures via blocks. However, unlike Smalltalk, Self allows the user to *override* the built-in control structures and observe the effects. Smalltalk shortcircuits the primitive control structures in the implementation (Smith and Ungar, 1995).

Method Activation A particularly intriguing language concept in Self is the way method activation is unified with prototypes. Recall that methods are regarded as a special type of *object*. In fact, methods are prototypes for so-called *activation records*: they are copied and invoked to run the code they encapsulate. This in contrast to class-based languages like Smalltalk where the method is said to *describe* the activation record (Ungar and Smith, 1987).

Whenever a slot containing a method object is queried for its contents, the activation record prototype is cloned and its code invoked. Before this code can be interpreted, the activation record's internals have to be adjusted to handle lexical scoping rules. First of all, the parent link of the clone must point to the receiver of the message. This will embed the method's scope in the receiver's. As mentioned before, this allows the method to access the slots of its surrounding (parent) objects. Second, the meaning of the `self` receiver has to be adjusted to incorporate local variables: messages sent to `self` are looked up in the method object first (as to access local variables), but the *receiver* of such messages is still the actual receiving object, not the method object. One can regard this semantics as the dual of the `super` keyword in most object-oriented languages, which starts method lookup in the parent, but keeps the receiver bound to the forwarding object (Ungar and

Smith, 1987).

Using these special semantics for `self` within an activation record, Self allows local "variables", instance "variables" and method lookup to be unified. They all follow the same ordinary method lookup semantics since they are all self sends. By making the activation record a child of the receiving object, one can interpret activation records as short-lived extensions of the receiver (Ungar and Smith, 1987). Note that Self has managed to unify variables and methods via slot access as follows: data objects merely return themselves upon slot access, while method objects run their enclosed code. However, this unification hinders the use of method objects as first-class entities: one cannot grab them directly (since grabbing a method would automatically run it) so we must resort to primitive objects, called *mirrors* which encapsulate some reflective operations on objects (Smith and Ungar, 1995).

2.5.1.3 Prototype-based Programming Techniques in Self

This section reports on a number of techniques or idioms on how to structure software in a prototype-based language such as Self. These idioms can be used to incorporate sharing between objects, to handle inheritance and to structure objects in a classless world. They show that we can organize our programs just as well without classes in a prototype-based language such as Self (Ungar et al., 1991).

Intra-Type sharing: traits To accomplish structured and reusable software, we require the ability to *share* state and behaviour among instances of the same type (or clones of the same prototype). Sharing enables you to change the behaviour of an entire set of objects with just a few strokes (Ungar and Smith, 1987). Without sharing, code would be duplicated, possibly leading to inconsistent objects whose code is not adjusted upon changes. Self solves this problem is through *parent sharing*: code is factored out by placing it in a separate object, which becomes the parent of all objects wishing to share the code (typically all clones of the same prototype). Such shared parents, usually containing the behaviour common to all its children, is called a *traits object*. It plays a role similar to a class in class-based languages (Ungar and Smith, 1987).

One might argue that introducing traits in prototype-based programs actually boils down to the reintroduction of classes. Thus, proponents of class-based languages have often argued that classes are necessary to structure programs. It can also be interpreted the other way around: prototype-based languages can readily express class-like features, yet they do not impose this structure and leave the programmer free to choose (De Meuter, 2004).

The reason why traits work in Self is because of late binding of self: behaviour (i.e. methods) that are not found in an object are searched for in its parent (the traits object). When the method of the traits object is then invoked, the `self` receiver variable is still pointing to the original (child) object. Thus, the traits method will access the correct "instance variables" of the original receiver. Let us rework the stack example from above using the traits technique:

```
stackTraits <- (|
  push: obj = (top: top+1. stack at: top Put: obj).
  pop = (top: top-1).
  getTop = (stack at: top)
|)

stackPrototype <- (|
  parent* = stackTraits.
  stack = array clone.
  top = 0
|)
```

We have factored out all behaviour into the traits object, which will be shared by all concrete stacks by means of a parent pointer. The advantage of this decomposition is twofold. First, stack behaviour is specified only once, leading to improved reusability, consistency and saved memory space. Second, by explicitly defining a prototypical stack, we can always use this prototype to clone new stacks. If the stack would have been coded without traits, it would quickly become confusing which stack would be a good (i.e. empty) stack to clone from. However, even with the traits technique, there is nothing that prevents users from changing the prototypical stack: this stack is in no way different from any of its subsequent clones. This scheme is thus sensitive to the prototype corruption problem (Blashek, 1994).

Traits are more flexible than classes regarding sharing. Traits are plain objects, which means they can contain both method objects (to support sharing of behaviour) and ordinary data objects, which then play a role similar to class variables in Smalltalk or Java. Traits objects generally do not contain information about the representation (state) of objects. This representation is embodied in a *prototype*, which is just an ordinary object, inheriting from the traits object. By cloning this prototype, we also shallow-clone its parent, which effectively makes the clone share the prototype's parent. The definition of a new data type in Self is thus spread over two objects: the "prototypical instance" of the type (containing state) and the shared traits object (containing behaviour) (Ungar et al., 1991).

Traits are thus more flexible since they make explicit what is usually implicit in class-based languages: objects instantiated from a class share their methods with siblings and contain their own data fields. By modelling such relation explicitly via traits, we can alter the scheme in a number of ways. We can for instance easily allow for just one object to override its shared behaviour (i.e. a method tied to only one object), singleton objects become extremely easy to implement: just fold the prototype and the traits into one single object. Abstract concepts can be implemented by providing traits but no prototypical instance: providing a representation is left to users of the traits object. The following paragraph will give even more evidence of the power of such explicit inheritance scheme.

Inter-Type sharing: refinement through delegation Since traits are just objects, there is no reason to disallow them from inheriting some behaviour themselves from other objects. Traits inheriting some functionality from other traits can be regarded as the equivalent of code sharing via subclassing in class-based languages. Generally, a new data type in a classless language can easily be derived from an existing data type by refining the corresponding traits objects (Ungar et al., 1991).

Some problems occur however, when we want to inherit both representation information (state) as well as behaviour. Since these concepts are explicitly kept in separate objects in Self, an object that wants to inherit both must do so by inheriting from two different objects. This poses no real problems in Self, since the language supports multiple inheritance. An object can thus declare a "data parent" and a "traits parent" (Ungar et al., 1991). Without multiple inheritance, this problem would manifest itself grievously. The explicit separation of both concepts would lead to the duplication of state description in the prototype of a new (refined) traits object: this prototype would have to explicitly incorporate all slots also present in the prototype of the original traits object [9]. We will come back to this problem when discussing the Pic% object model in section 2.5.3.3.

Using refinement through delegation, in combination with the traits technique again shows the gain in flexibility over class-based schemes. Class-based languages automatically extend the representation of subclass objects to also include the representation of the superclass objects. Using the above technique, it becomes possible to avoid such automatic extension and shun the representation of parental objects (Ungar et al., 1991). Consider for example a hierarchy of `Polygon` objects. A `Polygon` can be represented by including a list of its vertices. Thus, we have a traits object containing all operations defined over polygons, and a prototypical polygon object containing a list of vertices.

Imagine a `Rectangle` refinement of a polygon. We would like to represent rectangles by explicitly listing the four coordinates. In a class-based language, this is possible, but when making `Rectangle` a subclass of `Polygon`, we will automatically inherit the vertices instance variable. Using the traits technique, we can create a refinement of the polygon traits object, the rectangle traits. Furthermore, we can define a new prototype for rectangles, described using the four coordinates. This prototype inherits from the rectangle traits, but *not* from the prototypical polygon, thus, not inheriting the vertices. In class-based languages, avoiding automatic extensions can only be accomplished by factoring out common behaviour into an abstract superclass, which does not confine itself to a specific representation, but instead uses abstract methods to be filled in by the children. The key point in using the traits technique is that it is entirely transparent (i.e. visible and modifiable) to the programmer, which thus has complete control over his data type's representation (Ungar et al., 1991). All this is achieved in Self with a minimal number of

[9] In the Self 4.0 programming environment, a "copydown" mechanism can be used to achieve such incorporation. This circumvents the need for a data parent.

concepts.

Changing object behaviour Yet another example of Self's dynamic object model is the implementation of dynamic or changing behaviour. It occurs frequently that objects have to respond to messages differently, according to the state that they are in. Such behaviour is readily expressed by defining each state as a special subtype of the type under consideration. A prototypical object of this type would then inherit from one of the states. To change its state, it suffices to change its parent. This is possible in Self as a parent slot can be assigned to. Changing the traits object from which we inherit can be regarded as the equivalent of changing an object's class in a class-based language (Ungar et al., 1991). To illustrate the usefulness of such changing behaviour, consider figure 2.4. The figure depicts two traits objects: `anOpenedWindow` and `aMinimizedWindow`. When the object `myWindow` is minimized, all it has to do to change its behaviour with respect to future messages is to change its parent.

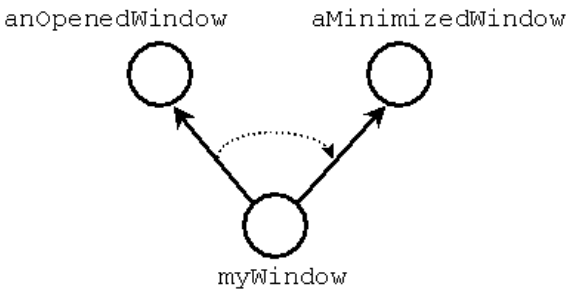

Figure 2.4: Changing Object Behaviour via Parent slot assignment

This is actually a very expressive implementation of the State Design Pattern described by Gamma et al. (1995) as a preferred way of structuring class-based programs in dealing with state changes. The State Design Pattern achieves such flexibility by promoting states to first-class objects. However, Self's solution is more expressive, since states are frequently singletons, trivially implemented in Self. Also, state objects usually require close coupling with their containing object. Some class-based languages cannot cope with this unless encapsulation is explicitly breached (i.e. having to provide accessor and mutator methods for the states to alter the object). Self avoids this problem since the state objects are parents, who are privileged to see their children's slots through late binding of self. Furthermore, state objects are usually shared. This is easily accomplished in Self, but requires more elaborate patterns such as Flyweight (Gamma et al., 1995) in a class-based setting.

Naming and categorizing objects If one does not want to get lost in the myriad of objects inhabiting a large Self application, some naming and categorization infrastructure must be created for the programmer to easily browse the program. For

example, it should be possible to refer to well known objects (such as `nil`, `true` and `false`) or user-defined data types, and to categorize name spaces as to avoid name clashes. The beauty of Self is that these features are incorporated without any extralingual support, it can be incorporated using only delegation and slots.

In class-based languages, programmers can typically refer to any class name from anywhere within the program (modulo import statements or visibility restrictions). In prototype-based languages, we are stuck with nameless objects. However, we *can* give them names by putting them into slots. The object's name is the name of the slot. Objects that exist solely as name providers for other objects are called *name space objects* (Ungar et al., 1991). Furthermore, if we can make the objects in our program *inherit* from such name space objects, then any named object can be retrieved just by sending a message to itself: the delegation-based scheme will find the named slot in the parent and retrieve the corresponding object. By putting name space objects in other name space objects' slots, we automatically get a name space hierarchy. In Self, the predefined variable `lobby` points to a global name space object containing references to objects made available by the system.

2.5.1.4 The Self Programming Environment

Programming in Self, just as in Smalltalk, is usually done via a sophisticated programming environment, providing the programmer with tools to efficiently structure his application. The environment also serves to hide the low-level object protocol from the user. For instance, subclassing in Smalltalk is usually accomplished visually with a few mouse clicks, but internally, a subclass is created by sending the `subclass:` message to the superclass.

The implementors of Self wanted to augment the interaction between the programmer and the Self world by creating a sense of *direct manipulation*. This is accomplished using two principles: *structural reification* and *live editing* (Smith and Ungar, 1995). The Self user interface is entirely built up of graphical objects called *morphs*. Structural reification implies that every graphical object is visible. There are no invisible "layout" objects. Furthermore, morphs can be arranged into a parent-child hierarchy. The semantics is that a child morph always "sticks" to a parent morph. This simple hierarchy thus provides ways to compose objects, without resorting to special container objects.

Structural reification also implies that any of these morphs can be grabbed and dissected by the programmer. The programmer can actually dissect his user interface, can clone parts of it, refine them, and reassemble them in his own application. This results in a very tangible and direct user interface (Smith and Ungar, 1995). The principle of live editing states that, at any time, an object can be modified by the user. Self provides *meta menus* to access any morph. Furthermore, the morph itself can be inspected and modified at the language level as a plain Self object.

Smith and Ungar (1995) regard the implementation of Self as the challenge to fool the user into believing in the reality of the language. They want to give

the programmer the impression that his machine is actually capable of containing a Self world. To this extent, they incorporate adaptive compilation techniques to improve response times (methods are first compiled fast and unoptimized, and are optimized adaptively if they are frequently used). They even de-optimize the code to allow the programmer to debug his program at the source code level, without noticing the use of advanced compilation techniques.

2.5.1.5 Conclusion

Self is a very pure object-oriented language, much in the spirit of Smalltalk. Its simplicity arises from its minimality: nearly everything, such as flow control, variable scoping, method invocation and primitive data types consists of just objects and messages (Smith and Ungar, 1995). To augment this simplicity, classes are abandoned for prototypes, which are simpler and allow for a more direct interaction. Self is probably one of the most mature prototype-based languages to date and has been used to prove that it is possible to organize programs without classes (Ungar et al., 1991).

2.5.2 The Agora Language Family

2.5.2.1 Introduction

Agora is a language developed at the Programming Technology Lab (PROG) of the *Vrije Universiteit Brussel* in the early nineties (Steyaert, 1994). It was initially designed as a very minimal object-oriented programming language. The idea was to start from objects and messages alone, gradually extending the language with new features on top of the basic model to study the semantic relations between the added features and the model (De Meuter, 1998). However, quite to their surprise, the language conceivers were able to add such features *without* having to change the basic model of objects and messages. Thus, Agora has shown that it is possible to add such features as inheritance, cloning and reflection without extending the language with concepts other than objects and messages (De Meuter, 2004).

Another goal of Agora was to combine the advantages of both class- and prototype-based languages, but to shun both their disadvantages. Thus, while some languages, such as Hybrid, try to take the *union* of the features included in both languages, Agora tries to strive toward an *intersection* of both paradigms (Steyaert and De Meuter, 1995). The reason why such an approach is needed is that in general, it is felt that prototype-based languages are too flexible, whereas class-based languages are too rigid. We want to capture some of the structuring mechanisms of classes in a prototype-based language, without sacrificing the flexibility of prototypes, which we have illustrated using the traits technique in Self.

This section will cover the most important concepts of the Agora language, with a particular coverage of the featured extension mechanisms. Next to that, we also discuss some problems solved by Agora's features and in what ways the

language has been a basis for our main prototype-based language under study: Pic%.

2.5.2.2 The Agora Language

We start our study of Agora by briefly summarizing its most important features, starting with the basic language values: messages and objects.

Messages Agora uses a syntax very similar to Smalltalk with respect to message sending. It distinguishes between two kinds of messages: ordinary messages and reifier messages (De Meuter, 1998). The main difference between the two lies in the evaluation order of the message arguments. Arguments passed to a method using reifier messages are not evaluated. This gives the same flexibility as the usage of blocks in Smalltalk and Self, as discussed previously. However, reifier messages are more expressive, since the sender of such messages does not manually have to "wrap" arguments in blocks. On the syntactic level, the distinction between ordinary and reifier messages is made by representing reifier messages using capitalized identifiers. On the semantic level, reifier messages can be compared to the so-called *special forms* of Lisp or Scheme (De Meuter, 1998).

Another distinction is made between *receiverless* and *receiverful* messages. Receiverless messages have a syntactically missing receiver object. Their semantic differences will be explained later on, when discussing public and private variables. As in Smalltalk, a final distinction is made between unary, operator and keyword messages. Operator messages include for example the message `3 + 4`, which sends the operator message `+` to the object 3. Keyword messages are used to pass arguments to methods. Agora features no less than twelve different types of messages, being ordinary or reifier, receiverless or receiverful and unary, operator or keyword (De Meuter, 1998). A few examples will clarify matters. The message:

```
x VARIABLE: 3
```

denotes a reifier, receiverful keyword message. It is used to declare a variable "x" in the object evaluating the message send. Methods are declared in a similar manner:

```
at:position put:value METHOD: (...)
```

A `METHOD:` message is sent to the receiverless ordinary keyword message expression `at:position put:value`, thereby creating a new method with the receiver as a formal pattern.

Objects There are two ways to create objects in Agora: either denote the object directly using a literal (such as a number, a string, ...), or create user-defined

objects *ex-nihilo* by listing a number of message expressions between square brackets, separated by semicolons. If one uses the `VARIABLE:` or `METHOD:` messages in such an expression, the object gets filled with either variable or method slots (Codenie et al., 1994).

Reifier!Messages Reifiers, just like special forms in Scheme, form the actual core of the language. Their evaluation entails very specific semantics. In Scheme, the language can be extended by "defining" new special forms using macro's. Likewise, Agora can be extended or adapted by implementing new reifier messages. Thus, Agora can be regarded as a language *family*, rather than just a single language (De Meuter, 1998). One important property of reifier methods is that they are *dynamically* scoped, in contrast with ordinary methods, which are lexically scoped. The context (or environment) in which a reifier is called is passed as a first argument to the reifier itself. This dynamic scoping makes sense, considering the fact that we will usually want to evaluate arguments in the context in which *they* were defined, not in the context in which the reifier is defined. In Scheme, the arguments to the `if` special form are also evaluated in the context where the `if` is used, not where the `if` construct is defined. In the rest of this section we will give a brief overview of Agora's more important reifiers.

As already mentioned, the `VARIABLE:` reifier installs slots into objects. More specifically, it installs two slots: a reader and a writer slot. The `METHOD:` reifier takes a formal pattern and a body and creates a new method in the object. The `CLONING:` reifier is more interesting: it takes an expression which will be evaluated within the context of a *clone* of the receiver. Thus, the `SELF` receiver variable used in the argument of a `CLONING:` message actually points to a clone and no longer to the original receiver.

Perhaps the most important reifiers introduced in Agora are `VIEW:` and `MIXIN:`. They allow for object extension and are in a sense Agora's surrogate for the addition or removal of slots. A view is also dubbed a *functional mixin-method*, while a mixin is actually an *imperative mixin-method*. A `VIEW:` message sent to an attribute with an expression creates a special type of method, whose body (the given expression) will be evaluated in an *extension* of the receiver upon invocation. This extension object is an independent object whose parent points to the original receiver (De Meuter, 1998). Consider the example:

```
point VARIABLE:
  [ x VARIABLE:0;
    y VARIABLE:0;
    circle:r VIEW:
    { radius VARIABLE:r;
      inCircle:p METHOD:
        {((p x) sqr + (p y) sqr) sqrt <= (SELF radius)}
    }
  ]
```

This example declares a simple `point` object, on which a circle view can be created by sending a `circle` message to the point. In response to such a message, a new object will be created, whose parent is the point. This object contains a variable `radius` and a method `inCircle`. Notice that the original point is in no way altered, hence the name *functional* mixin method. Consider what would happen if we had used `MIXIN:` instead of `VIEW:` in the previous example. In response to the `circle` message, the receiver would *destructively* be modified to include new variables and methods. Any other object (including any view!) pointing to the point will now *see* a circle. Mixins are thus very powerful abstractions that allow to change an entire object hierarchy in just a few strokes (De Meuter, 1998).

Just as in Smalltalk and Self, Agora defines its control structures in terms of message passing, rather than as a built-in language feature. Frequently used reifiers include `IFTRUE:IFFALSE:` and `WHILETRUE:`.

Local and public attributes Up until now, we have not yet mentioned ways to protect slots from external access. It is here that the distinction between receiverful and receiverless messages comes into play. Agora objects consist of a public and a private part. The private or local part can only be accessed by the object itself. This is accomplished through the use of receiverless message sends. Hence, messages sent to a receiver are always looked up in the public part of the receiver. Receiverless messages are looked up in the local part of the sender. Hence, sending an explicit message to `SELF` also implies lookup only in the public part of the object.

To distinguish between local and public slots, the unary reifier message `LOCAL` is used. `PUBLIC` can be used to explicitly make a slot public. If nothing is specified, `PUBLIC` is assumed. The difference between receiverful and receiverless message sends is analogous to the difference between message sends and function calls: receiverless message sends should be seen as function calls to functions in the lexical scope of the source code (De Meuter, 2004). Unfortunately, the interactions between local or public and receiverless or receiverfull messages together with nesting are far from trivial and render Agora very complicated.

2.5.2.3 The Scheme of Object-Orientation

Agora can in many ways be viewed as the Scheme of Object-Orientation (De Meuter, 1998). Just as everything is a first-class value in Scheme, everything is an object in Agora. Moreover, Scheme programs *are* Scheme data structures (lists). The same holds true for Agora: the parse-tree of a program is entirely represented by objects. We have already made the analogy between special forms in Scheme and reifier messages in Agora, which serve the same purposes. Furthermore, *apply* can be regarded as the fundamental operation in Scheme, whereas *send* (the sending of messages) can be regarded as fundamental to Agora.

This close resemblance with Scheme is also witnessed when we inspect the various components of both language interpreters. The Scheme interpreter needs

a memory of cons cells to store both data and programs and an *eval* procedure, parameterized by an environment parameter containing variable bindings. Furthermore, there is the fundamental *apply* procedure, which is the *only* operation defined on functions. It applies a function to a set of arguments, leading to the evaluation of its body in the appropriate environment.

The Agora interpreter is built up very similarly. It needs access to a memory of objects to store data and programs (parse trees). It also needs an *eval* procedure. In Agora, this procedure is replaced by a method, which is implemented by every parse tree object. Finally, there is one fundamental operation defined on each Agora object: *send*. This is where the difference between Agora and most other prototype-based languages can be observed: whereas Agora implements but *one* operation on its objects (send), other languages usually augment this interface (called the *meta object protocol*) with methods to add or remove slots, clone the object, etc.

2.5.2.4 Extreme Encapsulation

Agora has been claimed to "reintroduce safety in prototype-based languages" (De Meuter et al., 1996). Where then, was this safety lost in the first place? Prototype-based languages often suffer from what is called the *encapsulation problem*. Most prototype-based languages introduce a very small number of language values (e.g. only objects), but then introduce a myriad of "language operations" on those values, such as cloning, inheritance, adding slots to objects, etc. When objects are subject to change by such operators, they usually play no active role in the process: they are forced to undergo the changes enforced by the operator.

A typical example of this problem can be seen in Self, where – as noted previously – children can access their parent's slots. Since parent assignment is allowed in Self, any object can target any other object to become his parent, thereby gaining access to that object's data, without the parent object being actively involved. Similarly, if cloning in a prototype-based language is achieved via some `clone(obj)` operator, the object being cloned has no way of resisting the operator's effects. In such a language, it would be hard to express eg. a singleton object. If, however, cloning is achieved by sending a `clone()` message to an object, this can be seen as a "polite" request, which can either be accepted (default behaviour) or refused (by overriding the method) by the receiver.

Encapsulation problems involving inheritance or delegation are rather inherent in prototype-based languages, because of the late binding of self. By passing a different "self" to an object (for example during delegation), it becomes easy to fool an object (De Meuter, 2004). But even then so, the fact that any object (or class) can extend (subclass) any other object (class) can always lead to encapsulation breaches. Consider the following example, adapted from (Steyaert and De Meuter, 1995):

```
circleWithExpensiveGoldenWindow IS OBJECT
  private variable expensiveGoldenWindow;
```

```
  method drawInWindow(aWindow) : ...;
  method draw() : self.drawInWindow(expensiveGoldenWindow);

windowThief IS OBJECT
  method stealGoldenWindow(aCircle):
    return (aCircle extended with
                override method drawInWindow(aWindow):
                   return aWindow;
            ).draw();
```

Although this may seem a bit of a contrived example, it clearly shows that encapsulation breaches can be created even when children cannot access their parent's variables directly. In languages offering such modifiers as "protected", the situation is even worse if a class or object cannot control its own inheritors. The example illustrates that no object is safe if it can be exposed to uncontrolled inheritance. This is a serious issue in the context of distributed languages, where all code is not to be trusted equally. The breaching of encapsulation can lead to security leaks (De Meuter, 2004).

Agora's highly controlled inheritance mechanism through views and mixins solves such issues. Recall that the Agora computational model is completely based on objects and message passing alone. All other features, such as object extension, cloning and reflection are based on message passing. It is this simple message passing paradigm that validates Agora's safety claims. It allows for *controlled* inheritance, *controlled* cloning and *controlled* reflection. The inheritance scheme through views and mixins is called *encapsulated inheritance on objects* (De Meuter et al., 1996) and also *modular inheritance* (Lucas and Steyaert, 1994).

The main advantage of using views and mixins to define inheritance relations is the fact that the parent object is in control of everything. It is the receiver object that can allow or deny extension. Even better: it is the receiver object itself that can precisely determine how much it gets extended, since the extension code is entirely encapsulated within the object itself. There is no way to explicitly manipulate parent pointers. Likewise, cloning is supported in Agora only when the receiver has foreseen a `CLONING:` method. It is the receiver itself that decides whether and how it should be cloned. This is a solution to the prototype-corruption problem (De Meuter et al., 1996). The receiver is able to initialize the clone in the body of the cloning method, which allows us to model *constructor functions* from class-based languages.

This strong encapsulation and autonomy of objects has been coined *extreme encapsulation* by De Meuter (2004): objects should be subject to message passing and message passing alone. Any model in which one can circumvent message passing is potentially dangerous. Using only message passing, one has no choice but to actively involve the receiver in the process. Any Agora object knows *itself* unencapsulated (De Meuter et al., 1996), and has full power of extending or cloning itself, but a message passing client can only politely request the subject to adapt

itself.

2.5.2.5 Extension from the Outside

A frequently heard critique on the encapsulated inheritance technique outlined above is the fact that every possible extension to an object must be known in advance. Thus, we can only model *anticipated* extensions. If we want complete safety, this is probably the only satisfactory solution, any third party code *might* be malicious and destroy the object's encapsulation. There is, however, a technique to allow extensions "from the outside". The beauty of this scheme is that even this technique requires full cooperation of the subject.

Extension from the outside is achieved through a concept commonly known as *quoting*. In Scheme or Lisp, quoting is used frequently to represent programs as lists. This can be simply achieved by prepending a list with a quote. Agora allows for a similar mechanism through the `QUOTE` reifier message. Sending `QUOTE` to an expression evaluates to the parse tree of objects representing that expression. In other words, sending `QUOTE` reifies the underlying parse tree of Agora expressions (De Meuter, 1998). One can imagine quoting as *freezing* a given expression (i.e. capturing it without evaluating it). The expression can then later on be evaluated (possibly in a different environment). This corresponding *melting* process is achieved through the `UNQUOTE` reifier message, which evaluates the parse tree in the environment it was called in.

Where does this quoting mechanism lead us? Using parse trees as first-class objects gives us the ability to declare methods *outside* of an object, move them *inside* and then evaluating the parse tree within the object. Note that this scheme is *safe* from the point of view of the subject (the object under extension): to move the parse tree inside and to evaluate it requires its *active* cooperation. The following example will extend an object from the outside with a new slot:

```
colourSlot VARIABLE: (colour VARIABLE) QUOTE;
myCar VARIABLE:
  [ type VARIABLE: ...;
    wheels VARIABLE: ...;
    forward METHOD: ...;
    extend:slots VIEW: (slots UNQUOTE);
  ]

myColouredCar VARIABLE: (myCar extend: colourSlot);
```

The important point is again to notice that objects are still subject to extreme encapsulation. However, they *can* be extended if *they* allow to do so (De Meuter, 1998).

2.5.2.6 Reflection Protection

Many languages define strict access rights on objects, but allow these encapsulation barriers imposed on the base object to be circumvented on the meta level. By going to the meta level, one usually gains access to a very rich Meta Object Protocol (MOP) where multiple operations are defined on meta objects. Using these operations, it becomes possible to e.g. extend objects with new slots, although this would have never been possible at the base level. Particular examples of the liberal modification of objects can be witnessed in Self with its `_AddSlots:` message, in Smalltalk with its `at:put:` message and in Java with the `java.lang.Reflect` API. De Meuter (2004) coins the term *reflection protection*, which means that one should not be able to access more on the meta level than one is able to access on the base level.

Since many languages use reflection *operators*, we are stuck again with the same problems already mentioned when introducing extreme encapsulation (p. 33). How does Agora guarantee reflection protection? Recall that the interface of an object on the meta level consists of just *one* method, namely `send`. That is, a meta object does not contain anything more than a base level object. Since the `send` is the hallmark of extreme encapsulation, objects know themselves protected at all times, *also* at the meta level. Even users at the meta level can only send messages to the object, albeit now using the `send` message.

One might argue that it is worthless of providing a meta level where one cannot do anything more than at the base level. Agora consistently maps every base level message onto a meta level message and the other way around. As such, every message sent by the evaluator itself (implementation messages) can be intercepted and reprogrammed by the Agora programmer (De Meuter, 1998). It also facilitates the symbiosis between Agora and its implementation language: objects in the implementation language can be reified into Agora. The Agora interpreter cannot distinguish between objects created in Agora and reified implementation level objects: all objects understand but one message: `send`, and this is enough to support the entire meta object protocol.

Going to the meta level often allows one to write more flexible programs. A nice parallel of Agora's meta level `send` message is Scheme's `apply` special form. Using `apply`, it is often possible to write more flexible function calls, especially since function arguments are reduced to regular data structures (lists). Still, flexible as it may be, Scheme's `apply` operator does *not* allow the programmer to all of a sudden "dissect" functions. Thus, the meta level abilities acquired by using `apply` do not allow the Scheme programmer to break any encapsulation barriers on functions.

2.5.2.7 Conclusion

Agora can indeed be regarded as an intersection of class- and prototype-based languages. The mechanism of encapsulated inheritance on objects is a marriage be-

tween the object model employed by class-based languages and the inheritance model used in existing object-based inheritance (De Meuter, 2004). Safety is regained by adhering to the concept of extreme encapsulation, *also* when it comes to building inheritance (delegation) hierarchies, and *also* at the meta level (dubbed reflection protection).

The next section will introduce Pic%, which builds upon the features established by Agora. In fact, it addresses some of the issues not directly solved by Agora, such as the fact that it cannot directly cope with abstract entities. More importantly, Agora suffers from a re-entrancy problem. It cannot share methods the way Self can through traits, since Agora does not support multiple inheritance. This stems from the fact that the hierarchy for code-reuse is often perpendicular to the hierarchy to be modelled (De Meuter, 2004).

2.5.3 A Prototype-based Extension of Pico: Pic%

2.5.3.1 Introduction

The language we will be using for our experiments is called Pic%[10]. It is a direct descendant of the language Pico, designed just like Agora at the Programming Technology Lab of the *Vrije Universiteit Brussel* (D'Hondt, 1996). Although Pic% relies heavily on the concepts introduced in Pico, it borrows some fundamental ideas from Agora as well. In this section, we will first introduce the core language Pico. We will then proceed and discuss the extensions made to Pico to arrive at Pic%. This basically comes down to discussing Pic%'s "object model": how objects are represented and how they are affected by the language constructs. Finally, we will show that Pic% preserves Agora's claim to be an intersection between a prototype-based and a class-based language. It will also be shown that Pic% solves some problems that Agora could not.

2.5.3.2 The Pico Programming Language

Pico is a language that was originally designed to teach computer science to non-computer science students, such as mathematicians, biologists, physicists and chemists. As such, it was designed to be extremely small and *simple*, both from a conceptual point of view as well as from a syntactic point of view. To ensure conceptual simplicity, Pico's concepts resemble those found in Scheme (Abelson and Sussman, 1985). Although Scheme is extremely simple and expressive, its syntax is rather cumbersome to read, because of the consistent representation of programs as lists. To adhere to the second principle (easy-to-read syntax), Pico has mainly relied on syntax from calculus and algebra.

Pico Syntax Pico does not feature a syntax as regular as Scheme's. It also does not incorporate what is called "special forms": special "function names" that have

[10] % = o/o = object-oriented

a special status inside the evaluator. Yet, the language designers wanted Pico to be an extensible language, like Scheme, which can be extended through the use of macros and quoting (essentially adding one's own special forms). This problem was solved in Pico in an extremely simple way, without giving up the language's readable syntax (De Meuter et al., 1999).

Central to Pico is the notion of a dictionary (an *environment* in Scheme terminology). Dictionaries define a mapping from names to values. Pico's syntax is built around the manipulation of dictionaries, in combination with the values used during manipulation. Table 2.2 gives an overview of the basic syntax rules.

Invocation	Table	Function	Variable
Reference	`t[`exp_1`]`	`f(`exp_1`,...,`exp_n`)`	`x`
Definition	`t[`exp_1`]:`exp_2	`f(`exp_1`,...,`exp_n`):`exp_{n+1}	`x:`exp_2
Assignment	`t[`exp_1`]:=`exp_2	`f(`exp_1`,...,`exp_n`):=`exp_{n+1}	`x:=`exp_2

Table 2.2: Basic Pico Syntax

Names can be added (definition), modified (assignment) or retrieved (reference) from a dictionary. Moreover, invocations consist of tables (array), functions and variables. Pico stays close to Scheme in that *everything* is a first-class value: basic values, tables and functions can be passed as arguments to other functions, can be bound to variables and returned from functions. A notable difference with scheme is the lack of anonymous functions: Pico functions always carry a name (De Meuter et al., 1999).

Tables also extend the Scheme model of lists of cons-cells. Whereas cons-cells have a fixed size of 2, Pico's memory model is based upon variable-sized tables. Table indices range from 1 up to their declared size. When defining a table, the right-hand expression is evaluated for *each* entry in the table, allowing for expressive creation of e.g. upper-triangular matrices. Tables can also be created using the `tab` native, which takes any number of arguments and returns their values stored in a table. To give a general flavour of the Pico language, what follows is a standard definition of the Quicksort algorithm in Pico:

```
QuickSort(V,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 }) });
    if(Low < Right, QuickSort(V, Low, Right));
    if(High > Left, QuickSort(V, Left, High)) }
```

This example is built up entirely out of constructs explained in table 2.2, except for the curly braces. These braces are actually just syntactic sugar, translating $\{exp_1; \ldots; exp_n\}$ into a call to `begin(`$exp_1, \ldots, exp_n$`)`. `begin` is a function which evaluates its arguments and returns the value of exp_n. In Pico, argument evaluation is always left-to-right, in contrast to Scheme, which leaves the order unspecified. There are, however, some important extensions to the above table.

Beyond the basic syntax A first extension is the apply-operator `@`. Apply is used in conjunction with function definition to define functions that can take an arbitrary number of arguments. Consider the definition of a function `sum` taking an arbitrary number of arguments and returning their sum.

```
sum@args : {
  res: 0;
  for(i:1, i<=size(args), i:=i+1,
      res := res + args[i]);
  res
}
```

Using `@`, it becomes possible to represent the formal argument list of a function as a *first-class* value in the form of a table. The identifier following `@` will simply be bound to a table containing all actual arguments. The function can subsequently use the arguments simply by manipulating the table. Notice the strong similarity of `@` with Scheme's ability to represent arguments as plain lists. In Scheme, `sum` could be written as follows:

```
(define sum
  (lambda args
    (begin
      (define res 0)
      (map (lambda (arg)
             (set! res (+ res arg)))
           args)
      res)))
```

Armed with such a construct, it becomes possible to define `begin` in Pico itself:

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

`args` will again be bound to the table of actual arguments. This definition is correct due to three reasons. First, call-by-value parameter passing is used, so that all arguments to `begin` are evaluated. Second, the value of `begin` is always the value of the last expression. This definition ensures these semantics by returning the last argument stored in the table. Finally, and equally important, `begin` evaluates its arguments in the order they are specified. Since Pico *ensures* that argument evaluation happens from left to right, we can be confident that the above function behaves as expected.

A second native function that can be defined using `@` is `tab`. Recall that `tab` takes any number of arguments and returns a table of these argument's values. Yet this is exactly what `@` does internally, making `tab` breathtakingly simple to define:

```
tab@args : args
```

Completely analogous to Scheme, `@` can also be used symmetrically at a call-site. In such situations, its purpose is to make the *actual* arguments of a function application first-class. This is again accomplished by making them accessible as a table. Instead of using the canonical application notation `sum(1,2)`, the `sum` function can also be applied to a table of values using `sum@tab(1,2)`. In Scheme `(sum 1 2)` can be rewritten using a first-class argument list as `(apply sum (list 1 2))`.

Having to write tables using the native function `tab` can be cumbersome. Therefore, Pico also defines syntactic sugar for `tab`. The syntax $[exp_1, \ldots, exp_n]$ is translated by the parser to the application `tab(`$exp_1, \ldots, exp_n$`)`. This allows for an expressive in-line construction of tables. A similar trick is applied to operators. The Pico parser will parse `1+2` as the application `+(1,2)`. All these syntactic translations allow for simplifying the evaluation rules of the interpreter itself. Programmers can also define their own operators, and use them with an infix notation, which makes Pico programs very readable. All this can be achieved with enormous simplicity:

```
a # b : [a,b]
x << y: x * 2^y
!n : fac(n)
```

Lazy Argument Evaluation A second extension to Pico involves special parameter passing semantics. This is an extremely important part of the language, since it allows for the definition of control structures in Pico itself, thereby eliminating the need for special forms. When the formal parameters of a function are defined as variable references, the semantics is *call-by-value*, as in Scheme. However, when the formal parameter of a function takes the form of a function application, the resulting semantics are what is called *call-by-expression* in (De Meuter et al., 1999), reminiscent to Algol's *call-by-name*. Consider the following function definition:

```
f(g(x,y),z): g(1,2)+z
```

This defines a function f with two arguments: g and z. Whereas z is a normal variable reference, g is a function parameter. When f is called, its first parameter will be treated as the *body of a function*, which will be bound to g. Thus, the first argument will *not* be evaluated which results in *lazy evaluation* of arguments. When calling `f(x+y,3)`, a new function `g(x,y):x+y` is defined locally in the scope of f. The result of the call will be 6.

One should notice that the scope of g is the scope of f's caller. To f it looks like g is *dynamically* scoped. However, strictly speaking, all functions in Pico are *lexically scoped* and store their environment of definition. The environment of call-by-name functions is simply created at call-time. Their body will thus be evaluated in the environment of the caller. The resulting semantics are intuitive, considering the fact that the caller can then use variables visible in his scope and use them as "free variables" in the call-by-name function.

Armed with call-by-name parameter passing semantics, Pico can be turned into an extensible language, just like Scheme, Smalltalk and Self. the latter two employ *blocks* to achieve this. Pico is more expressive in this regard, since in Smalltalk or Self it is the caller of a function that must explicitly "thunkify" its arguments by wrapping them in a block, whereas in Pico it is the function itself that can enforce delayed evaluation (De Meuter et al., 1999). To prove the expressiveness of this parameter passing scheme, note that Pico's entire boolean system, together with its control structures are defined in Pico themselves. Booleans can be incorporated using *church booleans*, as featured in the λ-calculus:

```
true(t(), f()) : t()
false(t(), f()) : f()
if(cond, then(), else()) : cond(then(), else())
```

Without the `()` to delay the arguments to the `if`, such a definition of `if` would not be expressible in Pico, unless we make the client responsible for passing the arguments as function bodies.

Continuations In Pico, the run-time state of a program is a first-class entity. This means that it can be injected into the language value space. Pico inherits this powerful feature from Scheme, where such a value is known as a *continuation* (Abelson and Sussman, 1985). We will briefly explore continuations here, since we will be needing them later on in chapter 5. More details can be found in (Abelson and Sussman, 1985; Feeley, 1993).

A continuation actually denotes "all that remains to be computed" at the point where the continuation is captured. Continuations are best understood when looking at programs as nested expressions, each of which "returns its value" to the surrounding expression, when it gets evaluated. This "expression surrounding the value" is then the continuation of the expression yielding that value. In Scheme, a continuation is represented as a function taking the value as a single argument.

Capturing a continuation is accomplished using the function `call-with-current-continuation`, usually abbreviated `call/cc`:

```
(call-with-current-continuation
  (lambda (my-continuation)
    (... use my-continuation ...)))
```

`call/cc` takes a function with one argument as a parameter and immediately invokes this function, passing along its implicit continuation. The function can then explicitly apply this continuation to perform non-local jumps.

In Pico, the story is very analogous to Scheme. Pico defines a function `call`, having one parameter. This parameter is actually a call-by-name parameter for a function with one argument named `cont`. `call` will immediately evaluate its function argument, thereby passing the captured continuation (also called an *environment* in Pico[11]) to the function. An intuitive but incomplete definition could be:

```
call(exp(cont)) :: exp(<captured environment>)
```

The captured continuation can then later on be restored ("jumped to") via the `continue` native. Notice the major difference in *representation* of continuations between Scheme and Pico. Whereas Scheme represents continuations as functions taking one argument, Pico represents them as a separate `environment` data type. The consequence is that function application in Scheme becomes more obfuscated: if a continuation is applied, control does *not* return as in a regular function application. Pico's approach is more clear since it provides a separate native `continue`, explicitly denoting the program's jump. `call/cc` has many uses, among others the implementation of coroutines and the ability to define exceptions and exception handling mechanisms.

Meta-programming in Pico The craft of meta-programming consists of writing programs that are *about* programs. Compilers, interpreters, parsers and editors are all examples of meta-programs. To this extent, Pico offers a number of powerful features for the programmer to manipulate his language values. First of all, Pico introduces the three core evaluator functions `read`, `eval` and `print` for usage from within the language itself. Using `read`, one can convert a Pico program denoted as a string into a Pico parse tree, which is entirely first-class. Moreover, Pico has primitives (*natives* in Pico nomenclature) that can explicitly construct and decompose parse trees.

Perhaps the most beautiful meta-programming technique is the unification of Pico programs (values) with Pico data structures (tables). This is reminiscent of Scheme's ability to represent Scheme programs as Scheme data-structures (lists). Any compound Pico value, such as a table, a parse tree, a dictionary and even a

[11] not to be confused with Scheme *environments*, which are Pico *dictionaries*.

function can be decomposed by treating the value as a table. In fact, a first-class function *can* be treated as a table of size 4, consisting of a name, an argument list, a body and a lexical environment (dictionary). This opens up an enormous potential for meta-programs, since the treatment of functions as tables allows for access to the environment, which is impossible in Scheme. Dictionaries (Pico environments) can in turn be regarded as plain tables of size 3, consisting of a name, a corresponding value, and a "pointer" to the next dictionary.

Pico thus allows for *reflection*: the ability of programs to reason about themselves (De Meuter et al., 1999). Pico goes even further when combining its meta-programming facilities with its first-class continuations: it becomes possible to access the continuation, thereby being able to access the runtime stack of the program. Hence, a program can modify its own run-time state from within itself. This is an extremely powerful feature which is quite rare among programming languages.

Pico's reflection mechanism does not adhere to the reflection protection principle as introduced in section 2.5.2.6. The principle states that objects (values in this case, since Pico does not feature objects) find themselves encapsulated even at the meta level. This is clearly not so in Pico, as most values can be "dissected" as tables. Thus, the price that is payed for the reflection mechanism's flexibility is the lack of encapsulation.

2.5.3.3 The Pic% Object Model

In this section, we will gradually extend Pico with features necessary to transform it into the prototype-based language Pic%. This dissertation will be largely based on (D'Hondt and De Meuter, 2003).

Objects and messages The construction of the Pic% language starts by noting that environments are made first-class as dictionaries in Pico. Hence, they can act as objects containing simple slots: bindings with a slot name and a slot value (D'Hondt and De Meuter, 2003). In order to easily grab the environment of evaluation, a native function `capture` is added that captures the current environment and *reifies* it as a first-class value. Consider the following object:

```
counter(n): {
  incr(): n:=n+1;
  decr(): n:=n-1;
  capture()
}
```

Evaluation of an expression like `c:counter(10)` will bind c to an object, which is nothing more than the *call frame* of the `counter` function. In Pico, just as in Scheme, each function application creates a local extension of the lexical scope of that function. In this scope, parameters (`n` in this case) will be bound.

Any definition in the function body will be local to the function invocation and thus added to this call frame. `capture` will return this frame as the value of the entire expression. This kind of object-generating functions has been termed *constructor functions* (De Meuter et al., 2003b). Figure 2.5 depicts the object as seen by the evaluator.

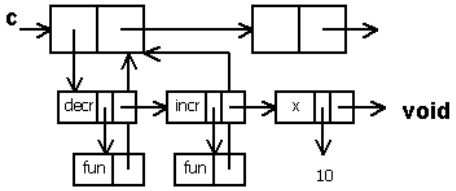

Figure 2.5: Simple Layout of a Pic% Object

Notice that such objects based on environments can also be created in Scheme, according to the same principles. However, since Scheme does not provide a `capture` native or first-class environments, the environment will have to be manually programmed, usually in the disguise of a "dispatch" function. This dispatcher, since it is defined local to an object, captures the "instance variables" of the "object" in its lexical scope. The mapping between names and values usually maintained by the environment must now also be programmed manually in the dispatcher. The `counter` object could then be written in Scheme as follows:

```
(define (counter n)
  (define (incr) (set! n (+ n 1)) n)
  (define (decr) (set! n (- n 1)) n)
  (define dispatch
     (lambda (msg)
       (cond ((eq? msg 'incr) (incr))
             ((eq? msg 'decr) (decr))
             (else (error msg " not found")))))
  dispatch)
```

To be able to access functions or variables within a particular object, name qualification is introduced (through the use of a dot-operator). The semantics thereof are to start the lookup in the qualified environment, instead of the current environment (D'Hondt and De Meuter, 2003).

Inheritance in Pic% Inheritance is added to Pic% by applying Agora's model of nested mixin methods as follows:

```
counter(n): {
  incr(): n:=n+1;
  decr(): n:=n-1;
```

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

Evaluating `p: c.protect(20)` will return an extension or view on the counter. This is similar to Agora's `VIEW:` reifier messages. In this regard, we can also view the `counter` method as a mixin-method on a global root object. Note the use of the "receiverless message send" (i.e. the dot without a left-hand expression) in the overridden `incr` and `decr` methods. They signify a simple super-send: messages are looked up in the parent.

It is clear that ordinary Pico functions are actually "promoted" to Pic% methods. The receiver of the message that will invoke these methods is simply the lexical environment in which that method exists. As for now, we will not distinguish between methods and functions. This implies that Pic% has first-class methods, a feature that is again very uncommon for a programming language. In languages like Smalltalk, Java or C++, methods are purely syntactic, they are not a language value.

Semantics have yet to be defined for function application as opposed to message sending. To this end, a native function `this()` is introduced which always returns the current receiver. This is reminiscent of Smalltalk's `self` and Java's `this` keyword. Function applications can then be interpreted as implicit messages sent to the dictionary denoted by the current scope. Late binding of self is introduced when performing super-sends: `this()` keeps pointing to the original receiver, even when executing a parent method via a super send.

Cloning Objects Another important operation in prototype-based languages is the cloning of objects. Here, the straightforward object model of unifying functions and methods runs into some difficulties. Recall that Pico is lexically scoped. Therefore, all methods of an object will have a link to their environment of definition, which is the object itself. This can readily be seen in figure 2.5.

Imagine cloning the `counter` object by sending it the `clone()` message. To achieve sharing of methods between c and its clone, they would need to be *shallow copied*, allowing the `incr` and `decr` functions to be used by both objects. However, since a function is always paired with a lexical environment in which to look up "free variables", methods will always only "see" the instance variables of their original object. In the case of the `incr` method, the free variable `n` will be lexically bound to the variable of the original `counter`. All clones of the `counter` would have their own variable `n`, but the methods `incr` and `decr` would only affect the

original prototype, as the lexical environment hard-wires variable lookup to take place in that object only. On the other hand, duplicating each method and adjusting the environment link would nullify sharing. The problem is shown graphically in figure 2.6.

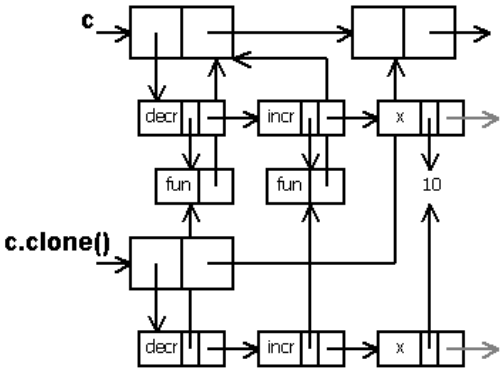

Figure 2.6: Interference of Lexical scope with cloning

The surprising fact about this problem is that there is a very elegant solution to the problem: abandoning static scope (D'Hondt and De Meuter, 2003). The alternative to static scoping, *dynamic scoping* implies that free variables will be looked up in the environment of *execution* rather than in the environment of *definition*. Consider the implications to the object model: methods will not carry a static environment. Hence, when they are *shared* between objects, their free variables (`n` in the case of `incr`) will be looked up in the environment in which they are executed, which is the correct object. It is this change in the semantics of functions that introduces the most fundamental differences between Pico and Pic%.

This elegant solution also immediately allows for variable overriding. Variable overriding is again a feature missing in most widespread object-oriented languages. It should not be confused with variable *shadowing* (D'Hondt and De Meuter, 2003). The main difference is that, when overriding a variable in a child, the method of a parent will see and use the child's variable, because of late binding of self. Variable overriding is possible since no identifier is hard-wired statically: all lookup is deferred until run-time. This is also the drawback of the scheme: method lookup (crucial to any interpreter) is less efficient when introducing dynamic scope.

Returning to our clone operation, how should it be implemented? Deep copying has already been ruled out since reentrancy is an absolute must: methods should be shared between clones. Therefore, methods were detached from their static scope. Yet entirely shallow copies of objects cannot be made either. This would result in the sharing of `n` for the counter object and the clones would not have their own state. Notice that this problem is similar to the one the Self group has solved via traits. There, behaviour and state are organized in different hierar-

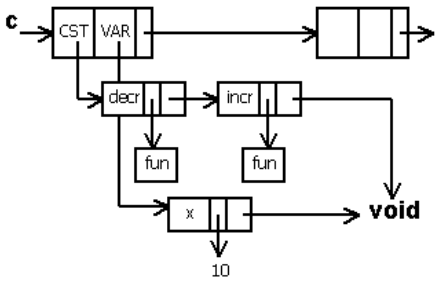

Figure 2.7: Pic% Object Layout Revisited

chies, and variable access is subsumed by message passing. Recall that the traits technique requires multiple inheritance to work when trying to build inheritance hierarchies with reentrancy. Agora also suffered from the reentrancy problem, and it was shown in (De Meuter, 2004) that the language offered no direct solution. So how does Pic% cope with the reentrancy problem?

The solution lies in the extension of the simple list environment model. To this end, note that shared code should be immutable. It is therefore necessary to introduce immutable variables – constants – into the environment (D'Hondt and De Meuter, 2003). Constants are subject to sharing upon cloning. Syntactically, there will be a distinction between *defining* variables and *declaring* constants. Constant declaration is the same as variable definition, except that double colons (`::`) are used instead of a single one. Figure 2.7 updates the view on a counter object, given that `incr` and `decr` are declared as constants.

The effect of a `clone` message on an object can now be defined. `clone()` will construct a new object, initialized with a *deep* copy of the variable part of `this()`, and a *shallow* copy of the constant part of `this()`. Figure 2.8 shows the counter and its relation with a clone. The constant part of an object can be regarded as a hidden traits object that does not interfere with the modelling hierarchy (De Meuter et al., 2003b). Notice that it is also possible to achieve sharing of other (constant) values, next to methods alone.

In Pic%, the semantics of `:` and `::` are overloaded. That is, they "mean more than one thing". We have already mentioned that `:` defines mutable variables and that `::` declares immutable constants. Furthermore, they also define encapsulation boundaries. Variables are *always* protected, while constants are *always* public. The rationale behind this is that variables usually constitute the representation, while constants are for the most part methods which export the public interface of an object. Message lookup always implies a lookup in the *constant* part of an object. Compare this with Agora where, as mentioned earlier, receiverful messages are also only looked up in the public part of an object. Table 2.3 summarizes the semantics of `:` and `::`. Summarized, one could state that $visibility = not(mutability)$.

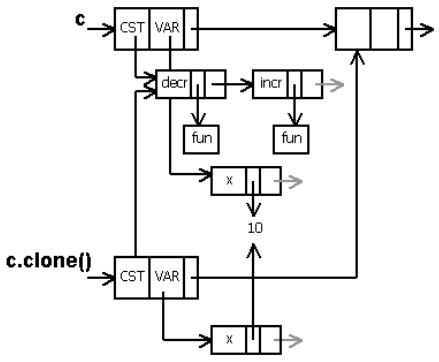

Figure 2.8: Effect of a cloning operation on a Pic% Object

		Mutability	**Visibility**
Definition	:	Mutable	Not Visible
Declaration	::	Not Mutable	Visible

Table 2.3: Definition and Declaration related to Visibility and Mutability

First-class methods and closures Although the reentrancy problem has been solved by adapting dynamic scoping rules for Pic%, there are still provisions required to handle first-class methods. That is, we must specify what happens when a method is "pulled out" of its object context as a first-class entity. This can happen, for example, in the following way (we assume `c` to be the `counter` object from previous examples):

```
{ method: c.incr;
  ...
  method();
  ... }
```

Notice that we do not *call* the increment method, but rather capture it and bind it outside of its initial context. Further on, the method is invoked. If we do not take precautions, then the method will probably fail because it cannot find `n` anymore. To solve this problem, a function being used in a first-class manner is automatically wrapped into a closure by the evaluator. That is, whenever the evaluator evaluates an expression to a function, it constructs a closure consisting of the function and the environment at that point in the execution. When a closure is later on applied as a regular function, first the enclosed environment is restored, and only then is the underlying function applied (D'Hondt and De Meuter, 2003).

One should note that the approach taken in Pic% to achieve first-class methods does *not* involve reflection, as is the case in Self where we *cannot* "grab" a method as we can in Pic%, since grabbing it would automatically apply it. To grab a method in Self, one has to ask the meta object (a so-called "mirror") of the object for the

method. Avoiding the meta-level to incorporate first-class methods keeps the Pic% object model simple.

2.5.3.4 Method Activation in Pic%

A particularly beautiful analogy can be drawn between Pic% and Self regarding Method Activation. In section 2.5.1, we have elaborated on Self's simple Method Activation scheme. There, we have shown that calling methods in Self has an elegant object-oriented interpretation. When closely studying (and implementing) Pic%'s call mechanism, we were struck by its resemblance with Self's mechanisms.

When a method (or a function) is applied, it will create a new call frame in which the method or function's arguments will be bound. This is also the frame in which local function variables will be defined. The frame's parent will point to the *currently active* environment. This is the key difference with Pico where the scope of a function application was an extension of the function's *lexical* environment instead of the current *dynamic* environment.

In Pic%, the call frame can be used to create *views* on objects: it is the return value of the `capture()` native. Since Pic%'s object model was unified with its environment model, this frame *is* an object! Thus, just as activation records in Self are first-class objects, so are Pic% frames. Because a message send causes the "current environment" to change to the receiver, the call frame will be an extension of this receiver. This is ideal for the scoping semantics: the variables of the receiver will be visible inside the method, which is exactly what the Self group has implemented by making the receiver a parent slot of the activation record.

Notice also that the Self group had to provide a special semantics for their `self` receiver within a method activation: message lookup started from within the method object, but the receiver still "points to" the original receiver. In Pic%, this is modelled by *always* making receiverless variable or function lookup start from within the environment denoted by `capture()`, but by making `this()` always point to the original receiver. Thus, the following method:

```
m(x) :: { ...x...; n(); ... };
```

can actually be interpreted as:

```
m(x) :: { ...capture().x...; capture.n(); ... };
```

except that the send to `capture` will leave `this()` unchanged. These semantics ensure that variables such as `x` are always found in the call frame (as a formal parameter) whenever this same variable would also occur as an instance variable of the receiver. The receiver variable itself could then be accessed using `this().x`. This close resemblance in unifying such complex operations as method invocation with object-oriented concepts between Self and Pic% shows that both languages strive for simple semantics.

We refer to appendix A.1 for a more formal treatment of method invocation and the entire Pic% object model.

2.5.3.5 An Intersection Between Classes and Prototypes

It has already been mentioned that Agora was designed to be an intersection between class- and prototype-based languages (Steyaert and De Meuter, 1995). Pic% has upheld this claim (De Meuter et al., 2003b). It can cope well with most of the disadvantages usually attributed to prototype-based languages. An overview:

- The fact that prototype-based programming languages could not ensure freshly initialized objects through a construction plan can be solved in Pic% by using the technique of the constructor functions. Each call to a constructor function creates a freshly initialized object. Beware that there is *no* sharing between objects successively created using constructor functions. Sharing is only accomplished when a clone is created of an existing objects that contains constant fields.

- The prototype-corruption problem states that a prototype, used by other objects to create a new prototypical instance, can be corrupted by accidental state changes. Subsequent clones will therefore also be corrupted. Pic% solves this problem in the same way Agora did: by turning `clone` from an operator into a message, extreme encapsulation is upheld: simple overriding of `clone` suffices to ensure correctly initialized objects.

- The fact that some concepts are inherently abstract, like a "stack" (one can only write code for "all" stacks, i.e. one has to code such a concept at the class-level of abstraction (De Meuter et al., 2003b)), forms no problems for Pic%. These concepts can easily be described using constructor functions.

- Perhaps the greatest problem was the lack of reentrance of methods. Code sharing is problematic since it can only be factored out into separate objects if we have multiple inheritance. Pic% offers a solution through the use of dynamic scoping and a clever object structure. The traits object is "hidden" and does not interfere with the hierarchy to be modelled.

Moreover, Pic% does not reintroduce the problems associated with class-based languages (such as per-object changes, ad-hoc singletons, lack of parent sharing and a more complex meta object protocol (De Meuter et al., 2003b)). Particularly interesting is the ease with which to enforce singletons in Pic%:

```
constructor(args): {
  ...;
  clone() :: this();
  constructor := capture()
}
```

Subsequent clone requests will be denied because we simply return the singleton, while the constructor function will only be able to run once: it overwrites itself with its created singleton object. This simple solution is only possible thanks to extreme encapsulation, an idea Pic% inherited from Agora. In short, Pic% has inherited cloning from prototype-based languages and constructor functions from the class-based world, and is thus a clean intersection between prototype-based and class-based languages (De Meuter et al., 2003b).

Parent Sharing Pic%'s inheritance model is a direct transposition of Agora's model, and it offers powerful abstractions, such as the fact that a parent can keep track of his own children. Likewise, because of extreme encapsulation, an object can keep track of its own clones. This feature can be used to simulate a simplified version of Kevo's *cloning families* (Taivalsaari, 1993):

```
object :: {
  myFamily :: prototypicalFamily.clone();
  clone() :: {
    myClone : .clone();
    myFamily.add(myClone);
    myClone
  }
  ...
  capture()
}
```

Since `myFamily` is a constant, it is shared between all clones. Moreover, *any* clone of the original object cannot escape being added to the family. Even if we send `clone()` to a clone of the object, the implementation will add the clone to the correct cloning family.

Reflection in Pic% The reflection model of Pico has been somewhat revised in Pic%. Instead of being able to "dissect" language values through tabulation (i.e. by interpreting them as tables), two native functions `get` and `set` are provided for exactly the same purposes. Using these native functions, any language value can still be inspected or modified. This means that Pic%, like Pico, does not feature reflection protection, as explained in 2.5.2.6. This can lead to nefarious security breaches in a distributed setup, where it is obviously not wanted that an untrusted remote object can freely inspect a bank account object. Building a safe reflection model for a distributed language is not considered any further in this dissertation.

2.5.3.6 Conclusion

Pic%, being an object-oriented extension of Pico, is a full-fledged prototype-based language. Yet its object model is extremely simple, thanks to the unification of

objects with the simple environment model. Pic% has also proven the existence of a stable object model with first-class methods (D'Hondt and De Meuter, 2003). Even more surprising is the fact that Pic% can cope with some severe problems usually associated with prototypes. No doubt, Agora's influence (extreme encapsulation and modular inheritance) has helped the language keep it simplicity *and* its expressivity.

2.6 Conclusion

In this chapter we have introduced prototype-based languages, both from a philosophical point of view and from a language-theoretic point of view. We have briefly explained the essential differences between prototype- and class-based languages, with respect to their mechanisms for empathy and templates. These differences all originate from the way the world is modelled. When we recognize that the world is indeed filled with distinct objects, which are subject to a similarity relation, prototypes emerge. These prototypes introduce their very own (programming) idioms. We have explained key concepts regarding these idioms, using the well-known Taxonomy of Prototype-based languages by Dony et al. (1992). When presenting this taxonomy we have devoted additional attention to the sharing mechanisms that prototype-based languages house, since this sharing will become an essential part of our dissertation.

We have also discussed three examples of prototype-based languages which have had a great influence on our own work, discussed in chapters 5 and 6. First of all we have introduced Self (Ungar and Smith, 1987), one of the most mature prototype-based languages available today. This language has a great influence since we also believe in Self's vision of simplicity through minimalism, which automatically leads to a strong affinity for classless systems. The second language introduced was Agora. The Agora language family, laying a number of important foundations for Pic%, addresses security concerns through *Extreme Encapsulation* and *Reflection Protection* (De Meuter, 2004). Finally we have introduced the language we will adapt in later chapters – Pic% – a minimal extension of Pico with prototype-based features.

In the following chapter we will focus on distribution, where we will among other things illustrate the role that prototype-based languages can play in such a context.

Chapter 3

Object-Oriented Concurrent Languages

3.1 Introduction

In the previous chapter we have introduced the realm of prototype-based languages. In this chapter we will discuss some issues related to the introduction of concurrency in a programming language, and more specifically how to introduce it in a language with only objects. These issues can be divided in two categories. The first category contains the somewhat more technical issues such as for example avoiding the pitfalls of shared data, whereas the other category is more oriented towards language design. Here we will for example need to choose how to support concurrency in our language.

We will begin our discussion by introducing both the actor model and the thread paradigm, which can be regarded as two extremes in the design space of concurrency models. Subsequently, in section 3.3, it is discussed what tools can be provided to guarantee that the data that is read and written concurrently is kept consistent. Furthermore this section will also devote some attention to the infamous "inheritance anomaly" (Matsuoka and Yonezawa, 1993).

Continuing, the difficulties of introducing concurrency in object-oriented languages will be discussed in section 3.4. These issues will be illustrated using the object-based concurrent language ABCL/1 (Yonezawa et al., 1986). Section 3.5 will then review a number of ways in which concurrency is tamed in various programming languages. Finally section 3.6 concludes our discussion on concurrency and links these concerns to distribution.

3.2 An Overview of Concurrency Models

This section addresses two "concurrency models" which we deem important enough to discuss because they present much of the foundations for our own concurrency

model of cPico, explained in chapter 5. The first is the actor model of computation, a very *functional* approach to concurrent programming. The other is the basic thread model, taking the stance of a more *imperative* view on concurrency. The "thread paradigm" will be explained using Java's concurrency model. Both models are introduced to expose the reader to two extremes on how to introduce concurrency into a programming language.

3.2.1 The Actor Model

The Actor model of computation (Agha, 1986, 1990) is a functional approach to concurrency. It is based on three main concepts: active objects, asynchronous message passing between such objects and behaviour replacement (Briot et al., 1998). For more details regarding the first two concepts, we refer to section 5.3. An actor is a self-contained independent component, having its own "thread of control". This means that an actor will autonomously react to messages by executing the "method body" itself. An actor consists of:

- An address and an associated **mail queue**, which will buffer incoming messages.
- A **thread** of control executing the methods of the actor.
- A **behaviour** or a script which denotes the set of methods and state variables of an actor.

Furthermore, the actor model is built upon three main primitives (Agha, 1990):

- A `create-actor` primitive which can dynamically spawn a new actor with a specified behaviour and a new mail queue.
- A `send` primitive which *asynchronously* sends a message from one actor to another. Asynchronous messages never return a (direct) result.
- A `become` primitive which allows for *behaviour replacement* as already noted in section 3.5.2.2. This primitive is very powerful and allows for an actor to change its state and methods. Note that a `become` only influences behaviour: it does not change the address or mail queue of the actor.

Agha (1990) draws the parallel between the actor primitives `create-actor` and `send` and the functional primitives `lambda` and `apply`. Functional languages do not incorporate an operation similar to `become`. This is to be expected as `become` is inherently an imperative operation and functional languages have no notion of mutable state. It is therefore this `become` operation that distinguishes the actor model from typical functional models.

Having introduced the context and constituent parts of the actor model, we can now describe the behaviour of an actor in response to a message. Whenever an

actor receives a message, the method in its behaviour should specify a replacement behaviour. This replacement behaviour will be used to process the *next* message in the queue. Since behaviours do not share state, processing of the next message may begin *as soon as the replacement behaviour is specified*. This allows for very elegant pipelined concurrency: processing of message $n + 1$ may begin while still processing message n. Figure 3.1 visualizes the notion of behaviour replacement. Note that an actor that always replaces its behaviour with its current behaviour is a purely functional actor: it never needs to "change state" (Agha, 1990).

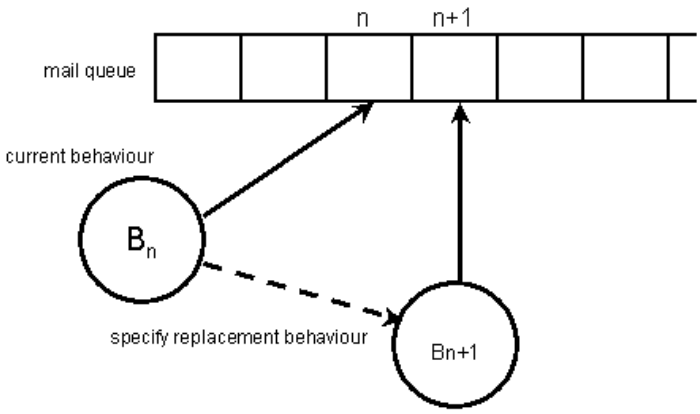

Figure 3.1: Behaviour Replacement in the Actor Model (Agha, 1990)

Because message passing is purely asynchronous, actors are unable to explicitly return results. Instead, a frequently used idiom in actor-based languages is to pass other actors as an extra argument to a message which is meant to "consume" the result of the message. Such actors are also called *continuation actors* (Lieberman, 1987) or *customers*. The problem with this idiom is that it forces the programmer to write his code in a *continuation-passing-style*, as explained in section 2.5.3.2. This causes code to quickly become scattered and unreadable and is perhaps one of the reasons why actor languages are not widely used in practice.

The use of customer actors is also used to *synchronize* actors. To express conditional synchronization, actors must resort to the techniques that will be outlined in section 3.5.2.5. Put briefly, an actor always returns the value of a method to some reply actor, usually called a *join continuation*. Such an actor will "consume" the returned value such that computation can proceed. This requires actor programs to be written in continuation-passing-style, however. By delaying the reply to such an actor, the "process" it represents is implicitly blocked, allowing for synchronization to be expressed. For a detailed example of an actor program making use of continuation actors, we refer to section 5.1.1.

To conclude, the actor model of computation is a clean, functional concurrent model. Actors are autonomous active objects, which process their own messages. They communicate asynchronously and therefore introduce a vast amount of po-

tential concurrency. Concurrency is controlled by explicitly specifying "continuations". Note that at no point have we mentioned classes, but neither have we mentioned delegation or cloning. Actors provide a low-level concurrency model on top of which abstractions can be built. Because of its object-centred nature, it seems only natural to try and use it as the basis for a concurrency model for a prototype-based language. The combination of the fundamental ideas from the actor model with the concepts of assignment and delegation is therefore interesting to study. Chapter 5 describes our reflections on the subject.

3.2.2 Threads and Monitors

This section will highlight a radically different approach to concurrency, one that is often found in class-based languages. Unlike the actor model, we cannot talk about *the* thread model. We refer to this model as a general term for any concurrency model in which computational entities called threads or processes are explicitly under the programmer's control, and where such entities communicate through shared mutable data. Whereas objects play a key role in actor systems, they are usually subordinate to threads in a thread model, where a thread "controls" multiple objects. Because there is no unified thread model, most programming languages employing threads have their own idiosyncracies. Rather than discussing threads at a more abstract level, we prefer to ground the discussion and will review the concurrency model of the highly popular Java programming language. First, we will introduce monitors as an early attempt of synchronization in a class-based context, on which the Java model is partly founded.

3.2.2.1 Monitors

Monitors were first introduced in (Hoare, 1974). The idea behind a monitor is to encapsulate mutable data that is shared between multiple threads by a data structure that mediates access to it. A monitor is thus a special kind of data type consisting of some private data and some *procedures operating on this data*. It is thus not surprising that Hoare immediately drew the parallel with Simula classes having private instance variables and methods operating on this data. The idea then, is to allow any thread to call a procedure on a monitor at any time, but at most *one* procedure can be executed simultaneously.

Hoare (1974) also introduces the concept of *condition variables*, as a method of achieving conditional synchronization (see section 3.5.2.1). Condition variables do not have an associated value. They can only be used in conjunction with two operations: `wait` and `signal`. Monitors can be efficiently implemented using Dijkstra's semaphores. To ensure that all procedures are mutually exclusive, a special semaphore called a *mutex* is associated with each monitor. Each procedure body must then be bracketed with corresponding P ("probeer") and V ("verhoog") semaphore operations on this mutex. In (Hoare, 1974) it is shown how monitors and condition variables can be implemented in terms of semaphores. Although

monitors allow for only rather low-level synchronization, it is one of the few basic concurrency constructs that has found widespread acceptance and is still used today in modern programming languages.

3.2.2.2 Concurrency in Java

The Java Concurrency model is based on three main concepts: threads, synchronization and monitors (Lea, 1999). Threads are a Java library class that encapsulate the well-known concept of a thread of control or autonomous process. Threads can share underlying file resources and objects constructed within the same program. They can be created dynamically by the JVM. A thread has to be explicitly *started* when created. A started thread will execute a special `run()` method that has to be either implemented by a subclass or by an aggregated `Runnable` object. A thread automatically stops executing whenever the `run` method returns.

Concerning synchronization, Java employs a lock-based scheme. Any object of class `Object` or its subclasses has an implicit lock associated with it. This lock can be acquired using `synchronized` blocks. The syntax is as follows:

```
synchronized (object) {
   ...
}
```

where the lock on `object` is acquired before the code in the block is executed. The lock is *automatically* released when returning (normally or through an exception) from the block. Methods can also be declared as `synchronized`, but this is considered equivalent to:

```
ReturnType method(args) {
   synchronized (this) {
      ... // method body
   }
}
```

Methods declared `synchronized` within the same class or superclass will thus always execute *mutually exclusive* within the object. Note that this exclusiveness does *not* guarantee atomicity: a method invocation on an object is only atomic if all methods are synchronized, since a non-synchronized method will not try to acquire the lock. Java locks are reentrant: recursive calls of a `synchronized` method do not lead to deadlock.

The final concurrency concept in Java are monitors based on the ideas by Hoare introduced above. Just as every object has an associated lock, it also has an associated queue or “wait set” which is entirely managed by the JVM. Programmers can manipulate this wait set using the methods `Object.wait`, `Object.notify` and `Object.notifyAll`. These methods are used to support conditional synchronization.

When a thread invokes a `wait` method on an object, it is automatically suspended and added in the object's wait set. A thread can only invoke `wait` on an object if it has previously acquired the object's lock. Upon executing `wait`, the lock is automatically released so that other threads can use the object again. Note that other locks held by the thread are *not* released. This might lead to hard-to-detect deadlocks, but is essential to guarantee correct semantics.

If a thread invokes `notify` on an object, an arbitrary suspended thread is selected from that object's wait set (if it is not empty) and awoken. This thread must then first acquire the lock on its object again. Since this lock will always still be held by the notifier, an awoken thread will always have to wait for the notifier to release the lock. This implies that, after a call to `wait`, the waiting thread should recheck its waiting conditions since other threads might have been using the object while it was trying to re-acquire the object lock. If the programmer does not take these subtle semantics into account, concurrency problems might arise. The `notifyAll` method has semantics similar to `notify` but awakes *all* waiting threads in the wait set of the receiver.

An example using Java threads is given in section 5.1.1. In conclusion, we can state that Java's concurrency model is based upon established and well-known concurrency abstractions (monitors), but is quite low-level in nature. Taking into account that Java is quite young a language, we find it surprising that Java has not employed more high-level schemes that integrate much better with the concepts of object-orientation. A similar remark is made by Briot et al. (1998) who state that Java favours a low degree of concurrency integration in the language itself.

3.3 Concurrency Issues

In this section we will elaborate on some important issues that pop up when designing concurrent programs. We will divide such issues in two broad categories.

First of all some issues address the problems of concurrent access on shared data, which are also called *race conditions*. The problems we therefore need to solve are also observed in the planning of database transactions, where data consistency is equally important. Whereas the problem space is identical (we need to maintain shared data correctly) the solution is not. In a programming language one does not have a specialized scheduler available. A set of low-level constructs should be offered to the programmer, which allow him to work out a solution that is best suited to his application needs. Especially in our context of distribution with lots of small clients other approaches might be too heavyweight.

The second aspect is more related to the fact that we are dealing with an object-oriented language for concurrency. As such we must take into account that a user will organize his programs using objects. In the object-oriented paradigm (possibly object-based) inheritance is important. Ideally we want to inherit the synchronization constructs as well. This is unfortunately not possible in the general case due to a phenomenon called the *inheritance anomaly* (Matsuoka and Yonezawa, 1993).

3.3.1 Race Conditions

Any concurrent programming language will at least offer a way to start concurrent computation. However this is far from sufficient in almost any context. If we are concerned with the actual correctness of a program's execution then a set of conditions must be observed. These conditions are closely tied to the correctness of data that may be accessed concurrently. This is why the problems that are discussed here are also observed when writing a realistic database management system which should feature transactional support. Traditional transactions should satisfy the ACID properties (Connolly and Begg, 1999a). We will first introduce the data consistency problems by means of two simple examples. Then we will evaluate how the ACID properties can be transposed in a programming language.

3.3.1.1 Incorrect Analysis and Ghost Writes

Incorrect Analysis and Ghost Write are two problems that are caused by unencapsulated access to shared data. To illustrate exactly the cases in which they occur we will illustrate their effect with the example of a simplistic Stock. We keep the set of operations as minimal as possible. First of all products can be bought, which reduces the number of products, and a new shipment can arrive which increases the number of products. Furthermore the stock manager can request for an inventory, to estimate based on the numbers of sold items and the available stock to order a shipment or not. The example is given in Pic%.

```
stock() :: {
  sold : 0;
  available : 0;
  bought : 0;

  buy(amount) :: {
    sold := sold + amount;
    available := available - amount};

  report() :: display( "Available : " , available, eoln,
                       "Sold : ", sold, eoln,
                       "Bought : ", bought, eoln);

  shipment(items) :: {
    available := available + items;
    bought := bought + items};

  capture() }
```

Though this code is quite simple, it can lead to unexpected results in a concurrent context, if the proper precautions are forgotten. Imagine that two threads

A and B make a customer buy a product concurrently. Both threads can then perform a lookup of `available` immediately after each other and thus find the same value, for example 5. This means that after this operation our number of available products is 4, instead of 3 which is what the user will expect. This is the traditional *lost update problem* (Connolly and Begg, 1999b) that is also encountered in database systems. The assignment of thread A is called a *ghost write* because the value it writes will never be read.

A similar problem exists with respect to reading where inconsistent information may be used due to the fact that some other method is executing at the same time. Consider the following scenario in our example:

1. The stock manager requests a report and at the same time a cashier registers that a customer has bought all 100 items in our stock.

2. The report function called by the stock manager performs a lookup of `available` and finds 100. The thread is then interrupted temporarily for some reason.

3. The cashiers thread registers the customers order of 100 products and finishes.

4. The report now continues and reports the number of items sold. However the sum of both will be more than what was bought.

This problem is also observed in databases and is named the *incorrect analysis* problem in (Connolly and Begg, 1999b). In the next section we will introduce the concept of *serializability* which will place restrictions on the sharing of data, in order to achieve correctness.

3.3.1.2 Atomicity and Serializability in Programming Languages

Serializability is another concept that is introduced in the context of database management. In such a context it is important to allow a maximal degree of concurrency, thus transactions must be able to work concurrently. Initially transactions can be separated based on the data that they access. However, this is not always enough. Through several techniques transactions can be scheduled such that their result is the same as what could have been obtained through serial execution of *some* permutation of the transactions. A serial execution means that all transactions are scheduled one after the other. Such a schedule which does interleave execution but only in an unobservable way is called a *serializable schedule*.

In the context of a programming language we can assume a more conservative stance. First of all the corpus of data that is encapsulated inside an object is usually a lot smaller than the data accessed by a typical transaction on a database. As such we can enforce the atomicity of an invocation in a much stronger way, by not allowing concurrent invocations on objects at all. Thus the notion of a transaction

and that of a method invocation can be merged into an *atomic invocation* (Briot et al., 1998).

The cPico model that will be introduced in chapter 5, will feature atomic invocations. Since transactions should satisfy the ACID properties, it may be of interest to see how these properties can be transposed to the context of atomic invocations. Any atomic invocation should be:

- ***A**tomic*: as the name *atomic* invocation already clearly indicates, this can be enforced in a programming language by only allowing a single method invocation to be active per object. In other words, intra-object concurrency is forbidden.
- ***C**onsistent*: atomic invocations do not leave the object in an inconsistent state. In a language which does not feature rules specifying pre- and post-conditions and invariants, this needs to be done by appealing to the responsibility of the programmer.
- ***I**solated*: atomic invocations are not impacted by other invocations that run concurrently. Since we explicitly forbid intra-object concurrency this property comes for free.
- ***D**urable*: which means that the result of an atomic invocation is not lost. However in our context we have left persistency concerns as a topic for future work.

The analogy between transactions and method invocation may seem a little far-fetched, because obviously differences exist. However, in section 4.5.2 we will discuss the language Argus (Liskov, 1988) which incorporates transactions as a language concept. In our own languages we will not support full-fledged transactions, yet we think that these rules can be used as a guideline to specify the characteristics of an atomic invocation.

3.3.2 The Inheritance Anomaly

When mixing concurrency with the object-oriented programming, we initially seem to gain quite a lot. Objects provide a good form of encapsulation, which is a good thing to have in a concurrent setting. Moreover, objects themselves can serve as a natural boundary to regulate concurrency by forbidding intra-object concurrency, meaning that multiple processes cannot be simultaneously active “inside” the same object. However, other synchronization concerns need to be addressed in the code of 1 object itself. The most typical example is that of a bounded buffer, which cannot execute a `get` command when it is empty. In a sequential context an error could be thrown since the fact that a `get` is performed on an empty queue is often a programming error. In a parallel context this in no longer true since we the state of a buffer that can be accessed by several processes cannot be predetermined.

This consistency must be maintained by the buffer itself, which should *delay* the evaluation of `get` until a `put` message arrives (Milicia and Sassone, 2004).

If synchronization code is tangled with functional code, all synchronization constraints will have to be completely rewritten when a method is overridden in a child object. This problem is termed the *inheritance anomaly* by Matsuoka and Yonezawa (1993). This problem has been studied extensively in the nineties, culminating in several proposals to reduce the problem. An overview of these solutions is given in (Matsuoka, 1993). We will first discuss two important criteria that should be taken into account when opting for a specific conditional synchronization construct, namely modularity and incremental modification.

Modularity is one of the essential features of such a construct if it aims to avoid the inheritance anomaly, since synchronization code that cannot be specified separately prevents us from reasoning about it in a consistent way. Thus without modularity, synchronization code for a given method will nearly always have to be rewritten. Two techniques that allow such modular specifications are discussed, being *guards* (Dijkstra, 1975) and *behaviour sets* (Kafura and Lee, 1989). Guards are boolean conditions that are evaluated to determine whether a method can be executed, whereas behaviour sets specify a set of methods that are applicable in the current state. They are discussed in more detail in sections 3.5.2.3 and 3.5.2.2.

Some consideration should be given to the fact that the inheritance anomaly itself is sometimes caused by the fact that behaviour is incrementally specified (e.g. a method invoking its overridden method using a super send). Thus, one should be able to do the same for the synchronization constraints. A set of operators should need to be defined that allow refining the guards or behaviour sets. Incremental modification is particularly interesting since it allows to reuse the synchronization for the parent which is still correct in case of a super-send, and allows to put additional constraints for the method on the child.

Frolund (1992) uses guards to specify when to *prevent* a method from being executed, because of the aforementioned observation that we usually want to impose additional constraints. When guards are used to specify when a method should be *available*, most likely this condition will be partially invalidated by new constraints. This would require a rewrite of the synchronization code altogether. However, if the guard *restricts* the use of a method, this guard can be reused in a subclass by imposing extra conditions using a simple conjunction of the guard of the superclass with new constraints.

Another approach is taken by Matsuoka (1993), who introduces among other constructs the notion of *accepted sets*. These sets specify which methods can be executed and thus in their most primitive form they are prone to inheritance anomaly. However in the model of Matsuoka (1993) they can be extended in a child, using a union construct. As such methods that are added can be made available for a child as well. However behaviour sets are not sufficient. Imagine we have a buffer with 3 states, `empty`, `intermediate` and `full`. Now if we introduce a `get2` function that gets 2 elements at a time. This would require us to split up the `intermediate` state and take into account the case where we have only 1 item

left[1].

Summarizing we must recognize that intertwining synchronization and functional code gives rise to the inheritance anomaly. In order to solve this important issue a construct is needed that is both modular and subject to incremental modification. Another important issue to consider is whether a conditional synchronization construct should state when a method is available or when it is *not* available. It seems that "disabling guards" are a good choice, though some problems are left unresolved (Frolund, 1992).

3.4 Adding Concurrency to an Object-Oriented Language

In the previous sections we have focussed on some of the more technical issues associated with programming concurrent programs. Some more high-level approaches to effectively designing proper concurrency constructs for object-oriented languages are now considered. This is not at all trivial, and different languages take different approaches. A good overview paper by Briot et al. (1998) categorizes these different approaches into three different but complementary categories. We will first briefly explain the *library* and the *reflective* approach, which we did not pursue. Later on we will present the *integrative* approach we adhered to. We will also use a set of guidelines set forward by Caromel (1990) which are concerned with minimizing the required changes for going from a sequential to a concurrent program.

The Library approach reuses the existing sequential idioms to structure concurrent programs. Thus the concepts in which the programmer expresses his algorithm are the same as before. However, a new set of classes or prototypes is provided to deal with concurrency. The objects that are offered range from *threads* over *semaphores* to possibly more higher level constructions such as shared queues which can be used to synchronize communication between two processes and *promises* or *futures*.

The Reflective approach strives for transparency, by bringing the concurrency issues to a meta-level. Thus the user can build abstractions for concurrent objects by supplying them with a specialized meta-object that can for example act as a monitor.

Our approach is called the Integrative approach. The goal here is to find a unification or coexistence for object-oriented concepts on one hand and concurrency concepts on the other hand. For example we can unify the notion of an object and that of a process into an *active object*. Furthermore there is also a parallel between a transaction and a method invocation, which can be merged into an *atomic invocation* which has the characteristics we have explored in section 3.3.1.2. Briot et al. (1998) identify one problem with this approach in a concurrent setting, namely the problem of *inheritance anomaly* which we have already discussed in section 3.3.2.

[1] This requires us to rewrite our `get` as well, since we should perform an additional check to see if we have one item left, and then become the new behaviour set.

3.4.1 Object-Based Concurrency Features

Caromel and Rebuffel (1993) specifies a set of language features considered essential when designing a concurrent object oriented language. Though they explore these features in the context of a class-based language, Eiffel//, at least some of these concerns will need to be taken into account in our model as well. We will summarize the features they discuss.

1. *Active objects* are a first concept that should be added since it provides a clean unification of processes and objects, as we have mentioned above. An active object is a good way to model a process, due to the inherently good encapsulation an object offers. Yet not every object should be treated as active. In Eiffel// this is done by only treating objects whose class inherits from a given class as active objects.

2. Consistent *asynchronous communication* between active objects is important to maximize the available parallelism in the system.

3. *Wait-by-necessity* is a technique which introduces return values for asynchronous objects. These objects are filled in when the value is available and processes block (wait) when they need the value. This construct allows one to introduce asynchronous communication in a language without need for adapting existing well-written software, i.e. there is no dependency on what a function does apart from the value that is returned. Moreover a return value allows for concurrent programs to be programmed in a much more natural way.

4. *Centralized control* means that rather than attaching synchronization control to a method directly, or even worse to weave it in the function code, we have a separate module where all concurrency control is expressed. The problem with explicit concurrency control remains that subclasses need to adapt the separate module.

5. *Automatic continuations* signify that if one method that needs to fulfill a promise `p1` calls another method asynchronously *and* tail-recursively that we should ensure that the intermediate method is not blocked and can already return. In our language this will be achieved through promise forwarding, which we will discuss in section 6.8.2.

6. Furthermore Caromel and Rebuffel (1993) strongly favours *explicit concurrency control* facilities which can be used to construct more declarative forms of implicit control. This topic is further elaborated in (Caromel, 1993). To support this decision he introduces *first-class methods*, *first-class requests*, and *access to the message queue*. Our investigation is currently centred around forms of implicit control.

7. *Sequential processes* signal that inside one active object only one thread is active. We believe that this choice is the most natural to make since there is no compelling reason to allow multi-threading inside one active object. Moreover single-threading avoids extra locking constructs and intellectual effort from the programmer.

To further illustrate how a concurrency model can be integrated in a language we will take the case of ABCL. Though we have similar opinions on a lot of essential issues, we refrain from using Eiffel//. In our context we think it is better to immediately look at an object-based language rather than sticking with a more traditional class-based language. The goal of this dissertation is to focus on concurrency and distribution issues in a prototype-based language. Furthermore ABCL also has a lot in common with our approach.

3.4.2 ABCL: an Integrative Object-based Approach

ABCL, the Actor Based Concurrent Language (Yonezawa et al., 1986) is an object-based concurrent programming language. It is heavily based on the actor paradigm, but introduces the important notion of state. It also introduces an additional "waiting state" for actors. It is never explicitly mentioned whether ABCL supports delegation with late binding of self. It does support cloning and class-like abstractions through constructor functions. Apart from that, ABCL is a true object-based language which has heavily influenced our own model explained in detail in chapter 5. We will now discuss the particular language variant ABCL/1, in particular how objects are modelled and how they communicate.

3.4.2.1 The ABCL/1 Object Model

In ABCL/1, all objects are *active*. That is, each object has an associated thread of computation and executes its own methods. See section 5.3.1 for a more elaborate discussion on active objects in our concurrency model for Pic%. An object can be in one of three states: dormant, active or waiting. It is initially dormant and becomes active whenever it receives a message. Whenever there are no more messages in its message queue, an object becomes dormant again. An object which is active can also block and wait for a certain message to arrive. It then transcends into waiting mode as long as the specified message does not arrive. This is very convenient for modelling conditional synchronization. Awaiting a message is achieved through a special `select` construct. This is called *selective message receipt* (Yonezawa et al., 1986) and is similar to Ada's `select` statement (Ichbiah et al., 1986). It is implemented without resorting to busy wait.

As already mentioned, ABCL is special in that it is an actor-based system introducing the notion of state into the otherwise functional actor paradigm. To protect this state from the concurrency problems previously introduced, objects with state cannot process more than one message at a time. ABCL/1 also allows for additional "constraints" to be attached to method patterns. The same method pattern

can occur with multiple different constraints. If multiple pattern-constraint pairs match an incoming message, the first one is selected. A typical bounded buffer is written down in ABCL as follows:

```
[object Buffer
 (state ...)
 (script
  (=> [:put obj]
      (if full?
       then (select
             (=> [:get] <execute get>)))
      <store obj>)
  (=> [:get]
      (if empty?
       then (select
             (=> [:put obj] <return obj>))
       else
        <remove obj from storage and return>)))]
```

Text between $<$ and $>$ is merely a summary of the source to focus on the more important concepts. An object consists of (optional) state and a script, which is a set of methods. Each method has a pattern (much like in Smalltalk). Conditional synchronization is achieved simply by waiting for a certain message to arrive that will add or remove elements from the queue, so that the blocked message can continue. This is an elegant synchronization scheme, yet hard to reuse since the `select` construct is embedded within the method body. This does not appear to be a problem in ABCL since it does not introduce any inheritance mechanisms to the best of our knowledge.

3.4.2.2 Message Passing Semantics

Messages sent to active objects are properly serialized by enqueuing them in a message queue (see section 5.3.2 for an analogy with our model). However, ABCL distinguishes between two kinds of "priorities" between messages: ordinary mode message passing and express mode message passing. In ordinary mode message passing, a message is simply enqueued in the message queue if the receiver is active. If the object is in waiting mode, waiting for some specific message to arrive, then a message is discarded if it does not match one of the required patterns.

Express mode messages can be accepted *even* when the receiver is active. It will then *interrupt* the current evaluation. This evaluation will be continued when the express message finishes, unless the latter explicitly aborts the computation. This type of message mode makes it possible to easily monitor an object's state or interrupt an object. Multiple express mode messages are enqueued in a separate *express* message queue. Allowing for such express messages introduces problems

with atomicity: no method invocation is guaranteed to be atomic anymore. The language designers have therefore added a primitive called `atomic` which will evaluate its expression without interruption.

ABCL is unique in the way message delivery between the caller and the callee is accomplished. The language provides no less than three types of messages. The rationale is to provide a *natural* means of synchronization between objects (Yonezawa et al., 1986), all easily expressible by the programmer. The three message types are explained in more detail below.

Past Type Message Passing This message type implies that the sender *does not* wait for any result to be returned. It just carries on its computation. This is a pure actor-like asynchronous method invocation. It is called "past type" because the call already finishes even before any action has taken place.

Now Type Message Passing The now type is largely equivalent to a simple synchronous method invocation. That is, the sender will wait for a result to be returned. The main difference with a pure synchronous method or function call is that the result is not necessarily the "return value" of the method. That is, the callee may respond to the caller explicitly and may still carry out some computation after having sent back the result. Recursively calling a method through now type message passing causes deadlock.

Future Type Message Passing This type of message passing allows for asynchronous method calls that still return a result. They are the basis for our own model of "promises", discussed more extensively in section 5.3.3. The major difference with the approach presented in this dissertation is that futures are not transparent in ABCL. Future type messages are particularly handy whenever the caller expects a result but does not need it right away. It can then go on and perform some other useful computations before truly accessing the result. This special return value is what is called a *future object*. It can be queried for the "result to be computed".

In ABCL/1, futures are rather special when compared to futures in Multilisp (Halstead, Jr., 1985), promises in Argus (Liskov and Shrira, 1988) or promises in our own model (section 5.3.3). They act as queues in which a callee can enqueue *multiple* return values. The queue's contents can only be queried by the object that has performed the future type message send. Yonezawa et al. (1986) note that a program's concurrency is increased by the use of such futures, since caller and callee can both perform useful work and synchronize only *when necessary*. This concept has been termed "wait-by-necessity" by Caromel (1989). The language designers also note that futures allow for a more expressive alternative in dealing with return values than callbacks or customers as was necessary in the actor model. Such callbacks or customers break down the method of an object into several pieces which then become scattered throughout the code (Yonezawa et al., 1986). Futures keep the code together in one method body.

One final peculiarity of ABCL is that the "return address" is first-class. It is called the *reply destination*. Such a reply destination is always implicitly present in now type and future type message passing. It is also possible for the caller of a message to explicitly provide a reply destination (object) different than itself. The syntax `receiver <= message @ replydestination` is used to provide such reply destinations.

By making the return address explicit, it becomes possible to program in a continuation-based style. It also allows for the reduction of past, now and future type message sends to only past type. Details on this reduction can be found in (Yonezawa et al., 1986). Put briefly, a now type message can easily be reduced to a past type message immediately followed by a `select` construct that will make the caller block until a result message arrives. The callee will then explicitly send the result back to the caller, thereby reactivating it. Future type message passing can be expressed in terms of past type message passing, where the reply destination is modelled by a special queue object representing the future.

3.4.2.3 ABCL/f

ABCL/f (Taura et al., 1994) is a statically typed heir of ABCL/1 which allows for ordinary functions next to objects, in contrast to ABCL having only active objects. In ABCL/f, mutable shared objects must be encapsulated in so-called "concurrent objects". Just like in ABCL/1, method invocations on such objects are serialized and atomic. The language is termed ABCL/f because its basic invocation mechanism is based upon future type message passing. The other types are still available though. Taura et al. (1994) argue that futures allow for an ideal transition from a sequential call/return paradigm to a highly asynchronous one. Again, the remark is made that asynchronicity without futures is much harder to deal with. Futures are even more explicit in ABCL/f than they are in ABCL/1. One has to specify their static type, clearly distinguishing them from normal values. The language designers have explicitly chosen to do so for efficiency reasons. The language is clearly designed to allow for efficient parallel computing. The programmer can for example specify the "processor" where a computation has to take place. ABCL/f is less interesting for us to study. It also breaks with ABCL/1 by introducing classes, thereby no longer being purely object-based.

3.4.2.4 Conclusions

ABCL tries to model the world in terms of active objects communicating (and synchronizing) through various forms of message passing. It is thus only normal that it stresses message passing and defines a number of features normally lacking in an object-oriented language, such as first-class reply destination and express messages. The latter were intended for natural modelling of objects. It is natural that an object can be "interrupted" to make it do something more important. The drawback is that interruption gives up atomicity of method invocations.

Yonezawa et al. (1986) present two alternatives: a mail priority model and an explicit `check-express` primitive that checks whether new express messages have arrived and executes them if this is the case.

To conclude, let us review the two most important aspects of concurrency in ABCL: how it is created and how it is controlled. Concurrency is created by having multiple active objects being able to execute their methods autonomously. Past and future type message sends also cause subsequent parallelism. Synchronization is achieved by making method activation serialized using a FIFO message queue. An important synchronization mechanism is the `select` construct, allowing for conditional synchronization. Finally, now and future type message passing allow synchronization at a higher level, conceptually through the use of the `select` construct.

ABCL was important enough to review it in-depth because of its large influence on our own design of cPico, presented in chapter 5. We have built upon the very "actor like" active objects and the future type message send. One feature lacking in ABCL is inheritance or delegation, in contrast to our own approach, where this relation between objects will be exploited in a concurrent setting.

3.5 Conditional Synchronization

Conditional Synchronization allows for two processes to synchronize based on some arbitrary condition. It is usually distinguished from other types of synchronization that determine the synchronization points in advance. For example, when performing a synchronous method invocation, the calling process will automatically await the return value. The "condition" is fixed in this case. In (Briot et al., 1998), conditional synchronization is termed **behavioural synchronization**. This term is often used in the context of conditional synchronization at the method level in an object-oriented language. There, conditional synchronization can be seen as the fact that a certain "request" (an incoming message) cannot be processed at the moment. An example of such requests could be a `dequeue` message sent to an empty queue. Rather than throwing an error, it would make more sense to make the requesting process block until the request can be processed. This makes synchronization of method invocation between objects fully transparent (Briot et al., 1998).

In this section the necessity for conditional synchronization is first explained. We will also specify a number of criteria to which a good synchronization scheme should adhere. Next, an overview of several existing conditional synchronization schemes is given. Each approach is checked against the postulated criteria. The last subsection concludes on all approaches.

3.5.1 Evaluation Criteria

We will introduce conditional synchronization using the running example of a bounded buffer. The example is quite representative as it is often the case that concurrent processes communicate using a bounded buffer data structure. Moreover, a lot of conditional synchronization situations are often slightly modified versions of the basic bounded buffer problem. The idea of a bounded buffer is that it provides some finite storage in the form of a fixed size queue. There are processes which can store elements in the queue (the *producers*) and processes which will retrieve them (the *consumers*). The idea is to make producers wait whenever the buffer is full and to make consumers wait whenever it is empty. Throughout the following section, where we provide an overview of various ways of coding conditional synchronization, we will work with a slightly more simplified version of this problem. We will use a bounded buffer of size one, also sometimes termed a *cubbyhole*. The cubbyhole can be modelled as an object having two methods: `get()` and `put(item)`. `get()` retrieves the item but should block whenever the cubbyhole is empty. `put(item)` stores an item and blocks if the cubbyhole is full.

We have evaluated each language construct supporting conditional synchronization according to the following three criteria.

Expressivity The language construct should allow for an expressive notation for the problem. Although expressivity lies in the eye of the beholder, one should always try and minimize the cognitive load on the programmer when dealing with synchronization. Expressivity can range from a very expressive declarative style to a more low-level operational style.

Reusability This criterion is nothing more than the demand that the language concept can gracefully handle the inheritance anomaly introduced in section 3.3.2. The conditional synchronization should be made as reusable as possible. Ideally, children of an object can reuse the conditional synchronization when they want to, but can modify it independently of non-synchronization behaviour if necessary.

Efficiency As always in programming language design, we should not add *inherently* inefficient language features. In the context of conditional synchronization, we can usually distinguish between efficient and inefficient schemes depending on whether or not synchronization is achieved through *busy wait*. Busy waiting implies that the synchronee is continually polling some condition until it evaluates to true. That is, the processes involved in synchronization waste precious processing cycles evaluating the condition over and over again. A conditional synchronization scheme avoiding busy wait is highly preferable.

3.5.2 Conditional Synchronization Schemes

This section will try and give the reader a general feeling of what different synchronization schemes have been used in the past to support conditional synchronization. Approaches vary in the above criteria and are usually coupled to certain concurrency models. We will illustrate each scheme with the cubbyhole example as explained above. We will also try to evaluate each scheme to the best of our knowledge.

3.5.2.1 Schemes Using Condition Variables

The notion of a condition variable is strongly related to the concept of a *semaphore*. That is, a process can *wait* on a certain condition variable until some other process *notifies* it. A condition variable has an associated implicit queue of suspended processes. Condition variables are found in many thread-based concurrent languages, among others Java (see section 3.2.2.2) and Obliq (Cardelli, 1994), introduced in section 4.5.4. Obliq incorporates explicit condition variables. The cubbyhole example is programmed in that language as follows:

```
let cubbyhole =
 (let notEmpty = condition();
      notFull = condition();
      var item = nil;
      { serialized,
        write =>
         meth(self, elem)
           watch notFull until item=nil end;
           item := elem;
           signal(notEmpty);
        end,
        read =>
         meth(self)
           watch notEmpty until item!=nil end;
           let hold = item;
           item := nil;
           signal(notFull);
           hold;
        end; });
```

Two condition variables, `notEmpty` and `notFull` are created, on which a method can be suspended through `watch` and resumed through `signal`. The `write` method will block whenever it finds the cubbyhole full. After having performed a write, the `write` method explicitly wakes up any blocked readers. The `read` method exhibits symmetrical behaviour.

Concerning expressivity, we think of this synchronization scheme as rather low-level. This is because it requires the programmer to think operationally. The programmer has to explicitly suspend processes and resume the ones it has suspended at appropriate times. Because of this operational approach, the necessary statements are often tangled with the normal code of the method. Thus, the method is also not reusable for inheritors. If the cubbyhole class or object would be extended, one would not be able to deal with synchronization in a modular way. Unsurprisingly, this synchronization scheme is extremely efficient. It suspends and resumes processes only when necessary. Moreover, condition variables can be readily implemented by semaphores or low-level test-and-set operations.

3.5.2.2 Behaviour Sets

Behaviour sets are based on the notion of behaviour replacement from the actor model (Agha, 1986, 1990). We will defer the discussion of this concurrent object-oriented model to section 3.2.1. For our purposes here, it is important to note that an actor has a behaviour, comprised of a number of methods. Actors can explicitly change their behaviour, thereby changing the amount and type of methods they offer to their clients. This can be used for purposes of intra-object concurrency and synchronization (Briot et al., 1998). One can also adapt this notion of behaviour replacement for the purposes of conditional synchronization, through what is called *interface control* in (Kafura and Lee, 1989). Indeed, the idea is that an object can be in a set of abstract *states*. Each state specifies a subset of the interface of the object as being "enabled". When an object is in a certain state, it can *only* process messages which are in this "enabled set". Other messages must wait for the behaviour to change in order to be processed.

The language ACT++ (Kafura, 1990; Kafura et al., 1993) is an actor-based extension of C++. ACT++ explicitly introduces Behaviour Sets for the purposes of conditional synchronization. The following is an example of the Cubbyhole in ACT++:

```
class Cubbyhole: Actor {
   int item;
   behavior:
     empty = {put};
     full = {get};
   public:
     Cubbyhole() {
        become(empty);
     }
     void put(int elt) {
        item := elt;
        become(full);
     }
```

```
    int get() {
       reply(item);
       become(empty);
    }
};
```

Note how the synchronization code has become almost completely transparent. This conditional synchronization approach is clearly very expressive, as long as the number of "abstract states" remain acceptable and the transitions between them are clear. Considering the inheritance anomaly, this synchronization scheme is much more amenable to reuse than using condition variables, especially if it is possible to override behaviour sets from within subclasses, such that new methods can be properly added. Sometimes, however, adding new methods will partition existing states into more fine-grained states. At that point, we are forced to override methods in the superclass merely for the sake of updating the arguments to the `become` statements, which is clearly an inheritance anomaly. An example is given in (Matsuoka and Yonezawa, 1993). Regarding efficiency, it is noted that behaviour sets can be implemented without resorting to busy waiting. Messages not eligible for execution can suspend their sender on a condition variable, such that they can be signalled whenever the behaviour is updated.

3.5.2.3 Guards

The concept of guarding statements with a boolean expression have been introduced to construct nondeterministic programs (Dijkstra, 1975). In such programs, one statement is chosen nondeterministically from a set of applicable statements for which the guard is true. They have been subsequently used in the context of concurrent programs by Hoare (1978). Guards allow for an extremely elegant synchronization mechanism. Of all considered synchronization schemes, they are the most expressive. In the context of object-oriented languages, a guard is almost always a boolean activation condition associated with a method. Example languages include Guide (Balter et al., 1994), having a `CONTROL` clause to group activation conditions, Eiffel's SCOOP extension (Meyer, 1993) where Eiffel's preconditions are turned into guards and cC++ (Surribas et al., 1996) having both precondition guards as well as "post-guards" which may block a method upon return. In a typical language employing guards, the cubbyhole could be written as follows:

```
class Cubbyhole {
   int item;
   public:
     int get() when (item!=null) {
        return item;
     }
     void put(int i) when (item==null) {
```

```
        item = i;
    }
};
```

Guards are extremely expressive because they are inherently *declarative*. It is plainly specified under what conditions (i.e. *when*) a method may be invoked, not *how* the different methods should interact in order to ensure these conditions. Considering reusability, guards are also extremely good in supporting reuse. The synchronization implicitly defined by the guard is completely separated from the functionality of the method. Thus, if guards can be updated separately from their methods, they allow for a good alleviation of the inheritance anomaly. This is confirmed in (Frolund, 1992), where it is argued that guards should be *incrementally modifiable* in subclasses, as is already possible with methods (see section 3.3.2).

Guards are usually defined as boolean functions of the instance variables of the object and the actual arguments of the method to which they are attached. Yet, their richness also forms their greatest disadvantage: they are hard to implement efficiently (Briot et al., 1998). A naive implementation that continually re-evaluates the boolean expression is not a viable option since this would imply busy wait. Most implementations will associate a condition variable with each guarded method. When a message has to wait because of an unsatisfied guard, messages can suspend themselves on this variable. At the end of *each* method in the class, the condition variables are signalled and all guards will have to be re-evaluated. This is still less efficient than previously mentioned techniques, but avoids busy waiting. This approach is taken in Guide, where the compiler will automatically generate these condition variables and "guard re-evaluation code" (Decouchant et al., 1988).

When looking at existing work in the context of guards, we find almost all approaches "attach" guards to methods, at the class-level. In a prototype-based language, we have the advantage of attaching guards on a per-object basis, just like we have the advantage of changing method behaviour on a per-object basis.

3.5.2.4 Chords

Chords are a relatively new type of conditional synchronization construct. They are derived from the Join Calculus (Fournet and Gonthier, 2002), and an implementation exists for an extension of the language C#, dubbed Polyphonic C# (Benton et al., 2002). Polyphonic C# adds two new concepts to the *very* Java-like concurrency model of C#: asynchronous methods and chords. An asynchronous method is defined as `async methodName(arguments)`. Such methods never return results or throw exceptions, and are executed by a separate thread, spawned upon message reception. The caller immediately returns from such asynchronous method calls.

A chord or "synchronization pattern" is a method whose header consists of a set of method declarations, separated by ampersands. The method body of a chord is *only* executed when *all* methods in the header have been called. "Partial calls"

to chords are implicitly enqueued until a matching chord becomes active. There can be at most *one* method which is not asynchronous in a chord. It is the caller of this unique synchronous method which will execute the method body when the chord becomes applicable. If there is no such synchronous method, the body will be executed in a new, separate thread. Continuing with the cubbyhole example, it can be expressed readily in Polyphonic C# as follows:

```
public class Cubbyhole {
  public Cubbyhole () {
     empty();
  }
  public void put(object o) & private async empty() {
     contains(o);
  }
  public object get() & private async contains(object o) {
     empty();
     return o;
  }
}
```

A `put` message will block if there is no corresponding `empty` message enqueued. If there is, `put` will send a `contains` message to the cubbyhole itself. Thanks to this message, another chord is activated which allows a `get` to proceed with exactly the value passed via the `contains` message. Notice that storage for the cubbyhole is allocated entirely implicitly by the "invisible" synchronization buffers.

We regard this method of conditional synchronization as being high-level and quite expressive. In (Benton et al., 2002) other examples are given of such concurrency abstractions as Rendez-Vous and Reader-Writer locks which remain expressive to formulate. This scheme has some problems with reusability however. Currently, Polyphonic C# takes a conservative approach: whenever a method is overridden, *all* methods associated with the overridden method through a chord must also be overridden. Why this is necessary is explained in (Benton et al., 2002), but it suffices to notice that by overriding methods defined in a chord, we could "break" the chord in the superclass. Because of the substitutability principle, clients of the original superclass could then deadlock. Considering efficiency, it is shown in (Benton et al., 2002) how the Polyphonic C# compiler translates Polyphonic C# classes into ordinary C# classes in terms of condition variables and queues.

3.5.2.5 Continuation-based Schemes

The final approach to conditional synchronization we will discuss is a less familiar and rather eccentric scheme. The idea is to represent processes by continuations

and having the ability to explicitly manipulate these continuations. If one can explicitly "tell" a continuation that it can "proceed" with a certain return value, then conditional synchronization can be performed simply by waiting for the appropriate moment to give the "go ahead" signal to (read: continue) some waiting process. We will illustrate how this is achieved in PScheme, a parallel dialect of Scheme and ACT1, an actor-based language.

PScheme PScheme (Yao and Goldberg, 1994) introduces parallelism in Scheme through six new special forms. Parallelism, synchronization and communication are all expressed through an extension of Scheme's *continuations* (see section 2.5.3.2) called *ports*. First of all, concurrency is easily created through a simple fork-join construct called a `pcall`. Consider a function f with three argument expressions a, b and c. Evaluating `(pcall f a b c)`, results in the evaluation of f, a, b and c in parallel. When *all* of the arguments are evaluated, f is applied to the evaluated arguments. Concurrency is thus introduced by forking over multiple arguments and synchronization is introduced by implicitly *joining* all parallel evaluations *before* calling f.

One can think of the `pcall` as creating three "links" between the function parameter f and the three arguments a, b and c. The evaluator then sends the values of a, b and c across these "links" to f when they are evaluated. f can only proceed when all three values have travelled over the links. Exactly these "links" are made first-class in PScheme. They are called "ports". Getting hold of a port is done exactly as is done with continuations using `call/cc`. PScheme introduces the special form `call/mp` (call-with-current-multi-port) to grab the "link" to which this computation's value should be sent. The value itself can then be explicitly "thrown" through the port to f using the `throw` special form (Yao and Goldberg, 1994).

The beauty of multi-ports in PScheme is that one can throw more than one result through a port. For example, the code evaluating a might send both values $v1$ and $v2$ through the port. The function f will then be applied for *each* triple of values for a, b and c sent through its incoming ports, allowing for elegant stream-based programming. Ports then act as queues separating consumers and producers implicitly. Next to a multi-port, PScheme also has single ports which only accept *one* value and discard any subsequent values thrown into them.

A small programming example in PScheme is shown below, adapted from (Yao and Goldberg, 1994). The example defines a simple recursive algorithm which traverses a binary tree in parallel, returning the first value found corresponding to a given key or `#f` if the element could not be found. Concurrency is created by passing the recursive calls as arguments to a `pcall`'ed function. The port through which the value should be sent is passed along, so that a found value can be thrown through the port using `throw`. The port itself is made available in the last line using `call/sp`. Notice that the port used here is a "single port", thus, only the value of the first `throw` will be used.

```
(define (find// elt tree port)
  (cond ((null? tree) #f)
        ((eq? elt (key tree))
         (throw port (value tree)) (die))
        (else (pcall (lambda (e1 e2) #f)
                     (find// elt (left tree) port)
                     (find// elt (right tree) port)))))

(define (find elt tree)
   (call/sp (lambda (port) (find// elt tree port))))
```

In (Yao and Goldberg, 1994) it is illustrated how the new special forms can be used to implement semaphores, rendez-vous synchronization and even futures. The key idea in using them for conditional synchronization is that a `pcall`'ed function will not execute until it receives all the necessary values from all input ports. A certain thread that must evaluate one of those arguments can "suspend" itself by capturing its port, passing the port to another port and then die. The other thread is then responsible to generate some appropriate value and to send it through the port of the lost thread. This way, the function that is still waiting to be executed can continue since the value it has been waiting for finally arrives. Although ports are a very powerful concurrency mechanism, Yao and Goldberg (1994) themselves admit that it is quite low-level and only meant to be used to build more high-level abstractions. Since port captures and throws are not modularly separated from other code, this synchronization scheme will also not be very reusable. Ports themselves can be implemented without suffering from busy wait.

Guardians in ACT1 The object-oriented concurrent language ACT1 (Lieberman, 1987) defines special *guardian actors* to incorporate conditional synchronization. Recall from section 3.2.1 that an actor continually receives messages and executes the associated methods itself. Sometimes, as we have noticed in the cubbyhole example, a message must be processed the result of which can not directly be returned. That is, when processing the `get` message on an empty cubbyhole, what useful result can be returned? Lieberman (1987) makes a similar remark: a reply might sometimes have to be delayed until a message sent by another actor arrives. Thus, it is sometimes desirable to delay replying to a message and allow other messages to be processed first.

ACT1 defines *guardian actors* to deal with this kind of behaviour. Actors are fully serialized: they can only process one message at a time. When a message arrives that cannot be replied to directly, a guardian actor can save "all means necessary to reply", continue receiving other messages and reply later on whenever he sees fit (Lieberman, 1987). "all means necessary to reply" is simply the object to which the reply should be sent, usually called a *continuation actor*, because it will "consume" the value of the method and uses it to evaluate "all that remains to be computed". An example of the use of such continuation actors is provided in

section 5.1.1. Guardian actors are merely special in ACT1 because they can grab this continuation actor explicitly. In normal actor methods, the continuation actor is implicit and the method return value is implicitly sent to this actor. Guardian actors have to explicitly perform a reply and therefore are capable of delaying this reply until the conditions are met. Regarding expressivity, reusability and efficiency, this continuation-based synchronization scheme is comparable to PScheme's ports.

3.5.3 Summary

We now summarize the various conditional synchronization schemes discussed briefly in this section. Table 3.1 lists all conditional synchronization schemes, together with remarks on expressiveness, reusability and efficiency. By providing a thorough overview of possible synchronization schemes, several approaches to synchronization have become visible. It is also important to note that there is not *the one* conditional synchronization scheme. Each scheme has some advantage of its own, but usually lacks some other qualities to be considered ideal in all cases. When trying to incorporate a conditional synchronization scheme into a language, choosing the right synchronization mechanism is not trivial.

Scheme	**Expressiveness**	**Reusability**	**Efficiency**
Condition Vars	Explicit sync	Tangled in code	Least overhead
Behaviour Sets	Implicit sync	Tangled in code	Efficiently implementable
Guards	Implicit sync	Modular	Most overhead
Chords	Implicit sync	Difficult to override	Efficiently implementable
Ports	Explicit sync	Tangled in code	Efficiently implementable

Table 3.1: Comparison of Conditional Synchronization schemes

3.6 Conclusions

In this chapter we have tried to provide an overview on several important issues in concurrency that are related to our approach of concurrency in a prototype-based language, discussed in chapter 5. We have started out with the study of the two most extreme approaches to deal with concurrent programming. The first – functional – approach was the actor paradigm (Agha, 1986), which will be an important inspiration for our own concurrent language cPico. However the imperative thread paradigm will prove to be equally important when we try to relax the stringent constraints that actors impose on programs.

In section 3.3, some of the traditional problems of concurrent programs such as *race conditions* were pointed out. In a more object-oriented context, the problems of writing reusable synchronization code due to the *inheritance anomaly* (Matsuoka and Yonezawa, 1993) were discussed. Subsequently, in section 3.4, we have provided both a methodology and a set of intentional guidelines to keep in mind

how concurrency should be introduced in an object-oriented language, and have taken the language ABCL as a specific case. We have concluded the chapter with an extensive overview of different conditional synchronization mechanisms. Each of these synchronization mechanisms was evaluated with respect to expressivity, reusability and efficiency.

It must be kept in mind that building a concurrent language is not the only goal of this dissertation. The concurrency model introduced in cPico should serve as a basis for a decent distribution model for prototype-based languages. Therefore, in the next chapter we will first indulge ourselves in the field of distributed programming languages, before returning to concurrency in the context of our own model.

Chapter 4

Distributed Programming Languages

4.1 Introduction

In the previous chapter we have already addressed the issues associated with concurrent languages. However, our scenario sketched in chapter 1, shows that our vision imposes other difficulties, that are currently dealt with in an ad-hoc way. A prototype-based distributed language might offer the programmer the tools to express these concerns more simply. We therefore set out to discover precisely these tools. This chapter will be dedicated to identifying what problems are related to the concept of distribution. Moreover, we will explore the world of distributed languages and discuss the ones we deem most interesting.

Difficulties regarding distribution are mostly associated with the fact that a program may span multiple *locations* or *nodes*. Referenced objects can then be either local or remote. A programming language should provide a set of concise rules that state what the proper semantics are for accessing and modifying such local or remote objects. Also, objects may potentially *move* from one location to another. This may happen, for example, when an object is passed as a parameter to a method invoked on a remote object. The unit of movement is questionable. It can range from a sole object to an entire program. A program may move in its entirety, in order to find more computing power, or to follow its user physically. Sole objects can be moved in order to query a remote database, for example. The language designer has to solve various problems. What is the granularity of movement? When do we move entities to another part of the network? How do we keep movement safe for both the entity that is moved and for the host that chooses to accept the traveller?

This chapter is organized as follows. In section 4.2, we will elaborate on some issues that arise when trying to incorporate a distribution model in a programming language. Section 4.3 continues with a discussion on the aspects of *Strong Mobility*, which will become one of the more important features of our distributed language. Section 4.4 validates our choice for prototypes by evaluating prototype-

based languages in a distributed setting. There, we will argue that prototype-based languages have some flexible mechanisms whose power should not be underestimated in a complex distributed environment. Section 4.5 is dedicated to the explanation of what we believe to be a number of interesting distributed languages. Finally, section 4.6 summarizes our discussion on distributed programming languages and opens the road to the introduction of our own model.

4.2 Issues in Distributed Programming Languages

This section presents a number of problems that sneak into programming languages the moment that they are extended to cope with distributed environments. Each problem requires solutions that usually need to be incorporated within the language itself. Some issues are rather high-level, while others are more technical in nature. We will highlight the most important ones, describe the problems involved and propose a number of solutions.

4.2.1 Administrative Domains and Mobile Computation

A major issue in the development of distributed languages intended for use across Wide Area Networks (WAN's) are what Cardelli calls *Administrative Domains* (Cardelli, 1998). In a typical Local Area Network (LAN), the network is governed by a single administrator, who usually protects his network from the outside world using *firewalls*, which are able to block incoming and outgoing network traffic. Such encapsulation barriers pose problems for distributed applications trying to communicate for example via Remote Method Invocation mechanisms. A distributed programming language is not the only type of application suffering from such firewall protection. Middleware solutions such as CORBA are known to have trouble with firewalls. One of the advantages of text-based protocols, like SOAP (Simple Object Access Protocol) (World Wide Web Consortium, 2003) is that they can circumvent the strong access policies of firewalls by using standard HTTP ports.

Cardelli argues that languages which want to deal explicitly with computations that span a WAN (and possible the entire world using the World Wide Web) will have to explicitly introduce the notion of such barriers. In (Cardelli, 1998), the analogy is made with political boundaries and tourists which must be able to ascertain their access rights through passports, visa and airplane tickets when wanting to cross such boundaries. Cardelli (1998) starts by stressing the *observable differences* between WANs and LANs. The first is of course the notion of mutually distrustful administrative domains, of which a program will need to be aware. Second, a WAN is more susceptible to bandwidth fluctuations and network partitioning. This may have a detrimental effect on performance, but more importantly, it will blur the distinction between a *failure* and long *delays*.

Cardelli (1998) argues that to cope with these fundamental changes, our computational model should change as well:

> *In moving from local-area networks to wide-area networks, the set of observables changes, and so does the computational model, the programming constructs, and the kind of programs one can write. The question of how to "program the Web" reduces to the question of how to program with the new set of observables provided by the Web.*

To this end, Cardelli (1998) stresses that a possible solution might be the field of *mobile computation*, in which a computation need not be bound to the same (physical or virtual) machine. Using mobile computations, it is possible to circumvent some of the problems introduced by WAN's. By annotating computations with several access rights, these rights could be checked upon migration, as such being able to move across administrative domains. Once the barrier is crossed, the computation can continue freely in its new domain. Furthermore, by being able to move computations, bandwidth fluctuation can be minimized by moving to better suited machines. Also, if a failure can be anticipated, it is possible to escape it by migrating elsewhere.

4.2.2 Safety

When developing distributed programs, safety is of paramount importance. This should not be confused with *security*, discussed in the next section. Safety is more related to fault-tolerance and tries to minimize the risks that an entire system breaks down because a small part of it fails. Safe languages try to "contain" the impact of an error or a programming bug. Safety can be addressed at several *levels* of a distributed application (Schougaard, 2003):

- At the communication level, a safe protocol that can perform eg. error correction and detection makes the overall application safer.
- At the level of a "computational environment", safety can be ensured by the use of hardware memory protection (eg. memory segmentation). A "computational environment" can either denote the Operating System, or a Virtual Machine, such as a language interpreter.
- At the language level, we can allow for safer applications by building *strongly* typed languages[1], restricting pointers (eg. by not making them first-class), by performing automatic garbage collection, thereby alleviating the need for manual (de)allocation of resources and array bounds checking. Thus, Java or Smalltalk can be regarded as much safer languages than eg. `C`.

[1] This does not necessarily imply *statically* typed languages!

4.2.3 Security

Safety is necessary, but not sufficient, to ensure *secure* applications (Schougaard, 2003). Using distributed applications and especially in combination with mobility, it becomes possible to run arbitrary code on other machines. This obviously raises some security concerns. From the point of view of the host (a "computational environment"), it should be questioned to what extent it can lend its resources to an immigrated computation. Also, to what extent can the host influence the immigrated code? The host should stay in control at all times. There will also be data private to the host that should not be accessible to visiting programs. From the point of view of an immigrating computation, equally important security questions arise: is it possible to entrust the new host with its valuable computations? There will also be data private to a program that the host should not be able to access (Vitek et al., 1997).

Thorn (1997) introduces four basic properties to which secure applications should adhere:

- Confidentiality implies that there is no leakage of private information.
- Integrity implies that private data should *not* be modifiable directly by unauthorized parties.
- Availability should be maximized, eg. through the use of replication of objects. In order to promote availability, a system should also ensure that it is not prone to *Denial of Service* (DOS) attacks. In such attacks, a system is severely flooded or overloaded with data or computation, such that it slows down or possibly even crashes. Schougaard (2003) stresses the necessity of robust applications: the invisible computers in everyday devices become visible when they break down, shattering all illusions upheld by its functionality. In the context of Ambient Intelligence, this is even more important since part of the vision states that the devices are seemingly *merged* in the environment of the user.
- Authenticity implies that one should be able to trust the identity of a communication partner.

In (Vitek et al., 1997), a similar treatment on security is given, but at a slightly more concrete level. A Threat Model is presented of the various threats that should be avoided when dealing with a distributed or mobile object system:

- A *Breach of Secrecy* implies that there can be direct access to the private state of some process or object.
- A *Breach of Integrity* implies that one computation can change the state of another by sending state changing messages to the other computation's objects. These state changing messages can be used to wreak havoc by malicious computations.

- *Masquerading* is what breaks the aforementioned Authenticity property: a computation presumes the identity of another computation, thereby tricking another computation into performing some service for it (eg. executing some code).

- *Denial of Service*, as explained above.

Vitek et al. (1997) propose some solutions in dealing with these problems. They also show by means of examples that languages like Java are susceptible to the above threats. One problem inherent in object-orientation is that Masquerading can be easily accomplished using polymorphism. Using subclassing and overriding, it is possible to adhere to the superclass' interface, yet implement radically different behaviour. Using the substitutability principle, this may lead to serious security threats, also in Java. This is for example the reason that critical Java classes (like `String`) are declared `final`: they cannot be subclassed and as such eliminate polymorphism.

(Schougaard, 2003) again considers solutions to these security problems at several levels:

- At the communication level, a *secure* protocol is of paramount importance to ensure the Confidentiality and Authenticity properties.

- The "computational environment" (eg. Operating System) can partly ensure Confidentiality by controlling access of computations, by controlling the locks they hold on data, by controlling which system resources they may use and to what extent, etc...

- At the language level, scope and access rules should be used to protect the interface of data items or objects.

Note that it is *exactly* this language level security policy that we have stressed so much in chapter 2. There, we have advocated that Extreme Encapsulation and Reflection Protection are just the kinds of mechanisms necessary to make prototypes more secure, for exactly the reasons of security in mobile object systems.

Vitek et al. (1997) take the stance that a secure programming system is *not* a system in which one can write secure applications, but rather a system in which one *cannot* write *insecure* applications. This is an even stronger claim regarding security. Extreme Encapsulation partly follows this idea, in the sense that one *cannot* access objects in any other way than to send them a message.

4.2.4 Referencing Remote Objects

A first concrete problem when trying to write secure distributed applications is how to be able to find and communicate with other processes. In other words, how can applications get a reference to objects living in a separate process space? One way is to use a centralized name server, in which any process can publish or register

an object under a given name. Other processes can then query such remote object references by their published name, *provided they know the name server*. The major flaw in this setup is the inherently centralized nature of name servers. The advantage is that, once an object reference or process reference is retrieved, the name server usually no longer plays any part in the communication. To minimize failures, the name server should be replicated on a number of different machines. Another drawback of the name server is that the name of an object is globally unique, and must be known to any other process that wants to use it in advance.

A different setup is used by JXTA[2], which is a set of open protocols that allow any device connected to a (wired or wireless) to communicate in a peer-to-peer manner. JXTA – mainly developed at Sun Microsystems – explicitly promotes decentralized communication and favours peer-to-peer over client-server setups. One of the protocols offered by JXTA is a discovery protocol (the *Peer Discovery Protocol*), which allows for broadcasting XML-described service data over the network. Other services (or processes) get to know about the existence of the broadcasting service in this way. Other protocols allow for opening communication channels between services and synchronizing multiple services.

In our vision of using distributed languages in the context of Ambient Intelligence, a decentralized approach is highly favourable, since we are targetting a dynamic network. Thinking back of the mental image of a person surrounded by a "processor cloud", communication between personal mobile devices will most likely happen through a wireless broadcasting mechanism. In that view, M2MI (Many-to-Many Invocation) (Kaminsky and Bischof, 2002) can be seen as a promising paradigm for programming distributed systems. The main idea behind M2MI is that objects can communicate through multicasts: a message is broadcast to a set of interested listeners. Currently, M2MI has been designed as an extension for Java, where objects can implement a certain interface. An M2MI invocation of a method on that interface will then broadcast the message to every object implementing the interface. This paradigm is designed to be used in wireless networks, in which broadcasting a signal is inherent anyway.

M2MI allows for messages to be sent via *handles*. A handle denotes an abstract set of objects to which a message will be broadcast. Objects wanting to be exposed to external M2MI messages are exported to the M2MI layer (Kaminsky and Bischof, 2002). The M2MI layer will automatically create dynamic proxies which will handle M2MI messages. In short, M2MI is a language interface on top of Java. M2MI messages are routed to other "M2MI-aware" objects using M2MP (Many-To-Many Protocol) which broadcasts the message via the wireless network. M2MI avoids the need for central servers, which are certainly unwanted when considering small portable devices, which should not be tied to some centralized server.

[2]http://www.jxta.org

4.2.5 Remote Method Invocation

Remote Method Invocation (RMI) is usually referred to as the object-oriented counterpart of *Remote Procedure Call* (RPC). RPC involves the invocation of a procedure in another program (possibly on another machine). RMI is usually thought of as "sending a message to a remote object".

Levy and Tempero (1991) give an overview of the basic differences between procedural RPC systems and object-oriented RMI systems, through the use of an example application. One of the main differences is the time at which the *binding* is made between caller and callee: in most procedural RPC systems, binding is resolved statically and does not change, while RMI binding is often done at runtime, to improve on flexibility (the caller or callee objects may move around freely to other nodes). Another notable distinction with classical RPC systems is that it is often applied in a client-server setup, where clients invoke the exported methods of some server object. This often leads to a setup where the server is rather "fat". It is as such less amenable to movement since it is referenced by many clients, expecting to find the server object at a certain location. In object-based schemes, every object can communicate with every other object through RMI, thus, every object can be seen as a lightweight server. As such, an object-based scheme can more readily describe true peer-to-peer communication (Levy and Tempero, 1991).

Two issues regarding RMI are highlighted in this section: how to represent such "remote objects", and what are the semantics of parameter passing when performing such a Remote Method Invocation? (Schougaard, 2003) identifies the following main steps during a RMI:

1. Get a reference to the remote object (as outlined in section 4.2.4).
2. Invoke a method on the remote object.
3. Transfer the arguments to the node of the receiver.
4. Receive and initiate the call and arguments at the receiver node. This also involves resolving the remote object handle to a real local object. Information about where to send a result (some abstract return address) – if any – must also be specified.
5. Invoke the method with the given arguments on the local object.
6. Return the method return value to the specified return address.

The following sections will go into some more detail about the object representation and the parameter passing semantics.

4.2.5.1 Representing Remote Objects

The usual method of representing remote objects is through *proxies*. A proxy is a surrogate or placeholder for the remote object. The use of proxies here is actually

just an instance of the more general proxy design pattern described in (Gamma et al., 1995). Proxies allow for intercepting any message that would otherwise be sent to the underlying remote objects. They give the language implementor a hook in which they can "fake" the message send by sending a request over the network, waiting for the result, and then returning this result as the return value of the proxy skeleton. The proxy is just used as a wrapper that forwards the message in a different format.

There are some problems in using proxies as surrogates for objects, however. The most prominent one is the loss of what is usually called *object identity*. Every object has an associated identity, which is usually used for operations like equality tests (`==` in Java). The problem of introducing proxies is that the proxy's identity is *not* equal to the original object. One solution to this is to use message sends to test for equality (e.g. Java's `equal` method, or overriding C++'s `operator==`). Yet, such practice no longer makes the use of proxies transparent and can introduce subtle bugs if the difference is overlooked by the programmer. If proxies are not made entirely transparent by the virtual machine, bugs are bound to be made. For example, downcasting a proxy would fail, whereas the downcast may have worked if it would have been performed on the real object. In general, any operator defined over objects, in which the object itself does not play a part, poses problems for proxies, since they cannot intervene. In a language that conforms to the Extreme Encapsulation principle, introduced in section 2.5.2.4, these problems can be avoided more easily. If operators are replaced by message sends, the programmer can allow proxies to intervene by allowing them to override certain methods, either by delegating to the original object or by implementing specialized behaviour.

In (Pratikakis et al., 2003), a static dependence analysis is incorporated into Java to follow the flow of proxies throughout the program, as to replace certain operations (such as equality testing) by code that correctly handles proxies. For example, if static analysis would show that `o` *might* contain a proxy at run-time, then upon encountering

```
(o == otherObject)
```

the following code could be generated instead:

```
((o instanceof Proxy) ?
    o.equals(otherObject) : (o == otherObject))
```

To be more precise, the analysis of proxies in (Pratikakis et al., 2003) is used to incorporate *transparent futures* in Java. Even though a proxy and its real object could implement the same interface, thereby making them interchangeable in any Java program that only uses the interface, operations like the ones outlined above impose serious problems when trying to incorporate proxies.

Not all distributed languages use proxies (Schougaard, 2003). The language Emerald (section 4.5.1) uses globally unique identifiers for its movable objects.

All objects that can possibly be referenced remotely have a name that serves as a unique identifier. Guaranteeing that an identifier is unique is not always trivial. Furthermore, the identifier should contain enough information to extract the location of the object (such remote object information is usually stored in the proxy object itself in proxy designs). In Emerald, this is solved by employing a protocol where processes talk to each other to find out where the object is located. A caching mechanism is used to minimize process communication (Levy and Tempero, 1991).

4.2.5.2 Parameter Passing Semantics

When calling a remote method, the arguments to this method have to be passed to the callee, which can reside on a different machine. Hence, the objects that are passed must somehow be transported over the network if *call-by-value* semantics are used. This usually means that we will have to *serialize* the object (also called *pickling* or *marshalling* the object). Serializing an object means flattening it, so that it can be sent over the network as a stream of bits. On the receiver side, this bitstream must be interpreted, and an object must be reconstructed from it by *deserializing* it. Such conversions are *not* trivial because objects can be composed in a graph structure (an object graph). There is nothing that prevents this graph from having cycles. As such, we must encode a cyclic graph into a bytestream.

When deserializing such a stream, care must be taken to preserve object identity: if two objects both point to an object before serialization and all three objects are serialized, then upon deserialization the two should still point to the same object, it should not have been duplicated. Serialization is often seen as a costly process. Philippsen and Haumacher (1999) report that in Java RMI, about 25% to 50% of the time of an RMI call is spent (de)serializing. Note that call-by-value implies the duplication of an object, as such, the callee may not have the most recent version of the object. For immutable objects, call-by-value is ideal, since no subsequent state changes can occur anyway (Levy and Tempero, 1991).

One way of avoiding these problems is using *call-by-reference* semantics. Using call-by-reference, a remote object handle is passed to the callee, pointing back to an object at the caller's site. Note that this handle is *not* a pointer as is done in the non-distributed variant of call-by-reference. Pointers are in general invalid across machine boundaries. Call-by-reference circumvents serialization of object graphs, but introduces another problem: whenever the argument is used in the remote method, it will give rise to subsequent Remote Method Invocations because the object stayed on the caller's site. This raises obvious performance penalties. The programming language Emerald (section 4.5.1) allows for more sophisticated parameter passing semantics: *call-by-move* in which an object is *moved* to the callee's site (as opposed to call-by-value which moves only a copy) and *call-by-visit* which moves the object to the callee's site only for the duration of the method call: the argument is moved back to the caller together with the result. Using call-by-value or call-by-move semantics is one of the ways to introduce Object

Mobility, which will be discussed separately in section 4.2.6.

A last issue in RMI is exception handling, which is complicated if RMI is made transparent. In languages such as Emerald, the distinction between local and remote method invocation is invisible. However, RMI calls can time out, their target object may become unavailable and their target host can crash independently of the calling host. All this is complicated when invoking remote methods transparently, because one cannot write failure handlers for *every* method call (Levy and Tempero, 1991). Section 4.2.8 will come back to this.

4.2.6 Object Mobility

One important issue in mobile object systems is how, when and which objects have to be moved throughout the system. The *how* is concerned with how the programmer can express object movement in the language. The *when* states that object movement may happen explicitly, triggered at the programmer's will or implicitly, eg. instructed by the run-time environment. Finally, the *which* is concerned with *what types* of objects can be moved: all objects or only some "exported" objects, made available by the user.

An equally important question is the granularity of movement: *how many* objects get to be moved from one node to another. Typically, objects are intertwined in a so-called *object graph*. Naively moving every object reachable from a given target object may lead to the movement of the entire object space! Conservative movement algorithms can eg. only move a single target object, and perhaps some "attached" objects next to it. This is the approach taken by Emerald. Other approaches might move a transitive closure of the object graph, which can then be pruned at a number of points. This is for instance possible in Java serialization using the `transient` modifier. Variable references declared as `transient` will not be followed during serialization. When reconstructing an object, such "cut-off" references are replaced by `null` pointers.

There are a number of reasons why one might want mobile objects:

- Mobile objects allow for optimized distributed computing. One can move an object to another node in the system, so that communication with that object now becomes local at the remote node (Schougaard, 2003).
- Mobile objects can avoid anticipated failures (Cardelli, 1998), and can move to other machines when it is known that the current host will be shutting down.
- Some mobile devices have limited storage space, by moving objects, one could perform some sort of load-balancing.
- Combined with Code mobility, an application together with its living objects can wholly migrate to another machine, thereby being able to follow the user around physically.

The question whether one should opt for explicit or implicit object movement, depends on the application at hand. One could figure out where objects best belong *through design* (i.e. it is known beforehand which objects are mobile and to what extent). Alternatively, mobility issues could be left to the run-time system, which can detect certain *patterns* in the interactions between objects. The virtual machine could then move objects automatically to minimize communication overhead. A combination of both might be most effective: applications should be designed with the mobility of certain parts kept in mind, yet, at run-time, local communication patterns may be optimized automatically (Schougaard, 2003). Object mobility is an important issue that will reappear later on when discussing a number of contemporary distributed languages (section 4.5).

4.2.7 Persistence and Transaction Management

An important subset of the typical applications that lend themselves naturally to distributed solutions (such as banking systems, reservation systems, ...) has a high need for persistence. Persistence means that object state is maintained throughout system crashes. Persistence and Transaction Management are thoroughly studied in the field of (distributed) Database Management Systems. Incorporating such features in a programming language is not trivial. Most contemporary languages do not incorporate persistence, rather, developers have to use an external DBMS to store their objects. A full treatment of these concepts is beyond the scope of this dissertation. It is, however, important to note that matters get more complicated in a distributed setting.

Distributed persistence can be problematic when the network becomes unavailable. Imagine a program is performing a computation, while suddenly being disconnected from the network. Later on, the network becomes available again (such a scenario is realistic in the case of small mobile devices using wireless communication). At this point, the results of the offline computation should be propagated to the user, instead of being thrown away because of some network failure (Schougaard, 2003). This implies a need for concepts that allow for *graceful* handling of disconnecting objects. In the context of this dissertation we have not more thoroughly explored these ideas. This was in part delegated as future work. We will present some of our ideas based on the use of *promises* in section 7.3.2.

Possible solutions and approaches to persistence are both covered in our discussion on both Emerald (section 4.5.1) and Argus (section 4.5.2).

4.2.8 Partial Failure Handling

Handling partial failures is essential to obtain reliable software. First of all it is important to note that a distributed program is in fact being handled by several processors. This greatly decreases the dependency on a single processor, yet it also introduces the problem of partial failure. What should happen if a processor that holds part of the program suddenly goes off-line or crashes. In this context it is

also important to remember that Cardelli (1998) already noted that in a WAN there is no observable difference between a failure and a long delay where the latter may occur very frequently.

One traditional way to deal with this type of errors is to work with timeouts or by providing alternate code in case of a network failure[3]. However this proves to be counterintuitive when striving for transparency as these types of constructs reveal which objects are local and which ones are not.

There are other viewpoints on failure handling. There is for example the transaction-based point of view where the "effects" of a method invocation remain invisible to other components until the method has finished successfully. We will illustrate such an approach, closely related to database management techniques, in section 4.5.2 in the discussion of the distributed language Argus. Another approach to reduce the vulnerability to system crashes or devices going out of range is to employ a replication strategy for critical objects. Maintaining such replicas, especially when they need to be kept consistent, requires a complex and distributed management system. The issues of failure handling are important in a distributed context. They fall outside the scope of this dissertation, however.

4.2.9 Distributed Garbage Collection

Distributed garbage collection is a serious issue in distributed programs. The problem statement remains basically the same as in a sequential language: how can one free the memory occupied by objects that are no longer referenced by the program ("garbage"). The problem itself, however, becomes a lot more difficult to solve in a distributed setting. This is because any node, be it conceptual or physical, can have an arbitrary amount of references to objects on any other node. With mobile objects and mobile computation this scattering of the data graph becomes even more problematic. It is essential for the correct semantics of a distributed program that no "dangling pointers" across the network are created. This means that a node should not reclaim objects that were no longer referenced *locally*, but were still referenced *remotely* by other nodes. Garbage collection removes a heavy burden from the programmer, even in non-distributed programs, by automating memory management on the heap. In a distributed context, where the "heap" can be accessed in parallel by distributed nodes in a non-deterministic way, manual management becomes practically unfeasible.

Several different schemes exist for garbage collection on a single processor. These schemes can be divided in two big families: the *reference counting* and the *tracing* approaches. Both schemes exist in distributed garbage collection as well. Norcross (2003) states that the open challenge in this field is to define a distributed garbage collector that satisfies following properties. A distributed garbage collector should be:

[3]One example of this approach are Emerald failure handlers which will be discussed in section 4.5.1.4

safe which means that no object is collected before it is released by all nodes.

complete meaning that all garbage will be reclaimed eventually.

scalable which means that the collector does not impose restrictions on the distributed system.

independent meaning that every node can perform a partial local collection.

The latter two properties are considered desirable, rather than essential in (Norcross, 2003, p. 5). Nevertheless scalability is really important in a language design context where the environment on which the programs are to be deployed can vary largely. In order to achieve scalability the collection process will need to be incremental and non-blocking. This means that the algorithm can proceed without global knowledge of the system, and that it does not require all nodes to be synchronized at some point. Furthermore it is difficult to underestimate the importance of independence: if no progress whatsoever can be made in garbage collection without cooperation of other nodes a large memory leak is the inevitable result if a process is suddenly disconnected, due to a system failure for example.

We will now very briefly introduce an example of both reference counting approaches and tracing approaches in a distributed context. Note that the garbage collectors are supported by a local collector which does the collection for all objects to which there are no remote references (i.e. these schemes reuse non-distributed garbage collectors at every node, and in addition define a suitable communication scheme between them to handle remote references).

Reference Counting schemes maintain, for every object that is referenced from the outside, how many pointers are currently pointing towards this object. This can be accomplished using indirect reference counting Piquer (1991). Such an approach will create a hierarchical tree of counts instead of maintaining a single reference count[4]. Each node in the tree contains two counts, the local reference count and the *children count*, which is the number of other processes it has provided with a reference. When both counts reach zero, a message is sent to this node's *parent* in the hierarchy, to signal that it may decrease its children count. Whereas this solution is certainly promising in a distributed context, it still suffers from reference counting's inherent inability to collect cyclic garbage.

Tracing approaches will clean the memory by follow pointers from a given root. Some examples of tracing collectors on a single processor are treated in the survey of Wilson (1992). One example of an incremental tracing collector in a distributed context is the distance based scheme of (Maheshwari and Liskov, 1995). This system keeps track of the number of network links that have to be followed to find an object starting from a persistent root. This distance becomes infinite once the last link to that object is cut, signalling that the object can be reclaimed. Nodes in an unreachable cycle will always remain pointed at but since the distance

[4]The interested reader can find a clear example of why maintaining only one count is prone to errors in (Norcross, 2003, p. 9)

is periodically reevaluated, this distance will increase without bound. Figure 4.1 illustrates the initial state of a reachable cycle, as well as how the distance increases over time. a's distance becomes $m + 2$ since it presumes to be still reachable through b. Through successive distance updates the distance of the cycles will be larger than a given threshold. If this occurs the system can decide to move the cycle to one node, where it can be collected by the local garbage collector.

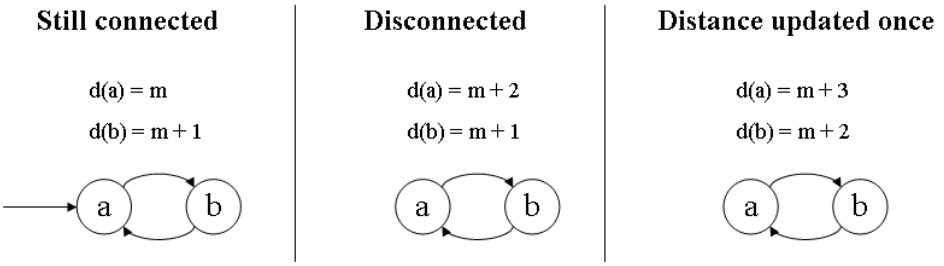

Figure 4.1: Cyclic garbage distance increases unbounded

4.3 Code Mobility

In our scenario in chapter 1 we have stressed the importance of movement. We have introduced Object Mobility in section 4.2.6 but if we look back at the scenario we can see that mere Object Mobility is insufficient to fulfill all the requirements we have put forward. The most obvious example is the running word processor moving to someone else's P-Com.

Here we can observe a need for Strong code mobility. This is related to Mobile Computation[5] which is concerned with how to write programs that are "aware" of their environment. This awareness can be observed by the user, as programs will sometimes move for several reasons. A compilation of reasons is presented in (Devalez, 2003). We will illustrate those that are related to the scenario we have presented in section 1.2.

Handheld computing is the most obvious reason to have strong code mobility. It is clear from our scenario that it is often useful to allow pieces of code to be sent to a handheld device, such as when the negotiator receives the running word processor of the secretary. The inverse is also interesting, as we see in the scenario when the P-Com of Maria beams the software demo to one of the secure hotel servers. This type of migration will often occur as handheld computing devices (especially if we consider cellular phones as part of the network) will often benefit from a delegation of their tasks to devices with more computing power. The language Borg, discussed in section 4.5.5, targets exactly this type of applications.

[5] which is concerned with software as opposed to the more hardware-oriented problems of mobile computing.

Other concerns are also reflected to a lesser extent in the scenario. The presentation that is given by Maria is obviously an *active document*. It contains much more information than present-day presentations. The decryption of the document is one of the things that is encoded in the presentation itself, as well as the interactive examples. *Remote device control* is observed when Maria requests the data projector to send her a controller process, to allow her to adjust the settings to her preferences. Finally also *Workflow management and cooperation* can be observed when the voting agent is sent to every negotiators P-Com, which allows him to vote anonymously, and yet only once.

4.3.1 Weak Versus Strong Code Mobility

The distinction between weak and strong mobility is made in (Fuggetta et al., 1998). *Weak Mobility* is code mobility in its most strict form, the code of an object is moved literally, and the object containing the code is regenerated on the other process. To be able to restore some state of computation, the object that is performing the computation should bring itself in some intermediate state which can be regenerated on the other node. This is the kind of mobility we currently see in for example Java applets (Devalez, 2003) and a lot of Java Mobile Agent frameworks, such as Aglets (Lange and Oshima, 1998). Here, the programmer is responsible to make sure that the Applet or Agent is in a consistent state, with all data that is relevant to its computation written out in instance variables. In doing so the reconstructed object will then be able to continue from a given point determined by the programmer. This is strictly less powerful than *Strong Mobility* in terms of expressiveness.

Strong mobility allows for migration of both code and the current execution state for an object. This means that the object can be frozen, along with the runtime stack at any point, without explicit coding by the programmer. The object is then shipped across the network, and re-activated there. Once this activation is complete the object can resume its computation without even noticing that it has moved. This type of movement is completely transparent, and avoids doing work twice. Especially when dealing with operations that may have side-effects this is an important advantage, since the (de)activation code of an agent is often error-prone. Weak mobility makes the programmer responsible for safeguarding variable consistency upon movement.

4.3.2 Advantages of Strong Code Mobility

Strong Code Mobility has several advantages on the implementation level, which are also described in (Thorn, 1997). First of all it increases the global efficiency. Network calls are most of the time very expensive, especially on low-bandwidth networks such as the ones that are used nowadays to connect PDA's. In such cases it is usually better to batch a whole set of calls, together with the computation that needs to be done between the calls in some sort of agent that will perform the

requested behaviour. The agent can be sent to the other node using only one call, so that it can perform its batched calls locally. This saves processing time since the cost of a remote call in comparison with a local call is not to be underestimated.

Furthermore strong code mobility also offers a benefit in simplicity and flexibility. Imagine an application server from which the employees of the company "download" their programs when they start to work. Updating and maintaining the programs is then easy, since a change on the server will automatically be propagated to all workstations. This may seem a bit odd, with the current day trends of not having workstations connected to a server, but instead independent computers connected by a network. However even in this setup Strong Code Mobility can simplify the maintenance problem. One could simply write an agent which performs the update and then allows it to traverse the intranet and perform the update locally (Devalez, 2003).

In the context of lightweight nodes in our network, Strong Code Mobility helps to alleviate the lack of storage space and computing power. If a process is too demanding to be executed on a small node, using Strong Code Mobility it can be moved to a more potent processing device without the penalty of having to restart some computation, which may have been going on for a long time. Also, running space-consuming applications can be temporarily moved to the application server, freeing space for other applications which the user needs more urgently.

Strong Code Mobility also helps to achieve Mobile Computing, when hardware is moving through some pervasive[6] and ubiquitous[7] network, it is useful that the software we write is also mobile, to follow the user on his walk through this processor cloud. Note that Strong Mobility is not inherently more powerful than Weak Mobility. That is, we can use Weak Mobility to solve any problem that can be solved using Strong Mobility. When considering expressiveness however, Strong Mobility outperforms Weak Mobility and allows the programmer to think at a more abstract level.

4.4 Evaluation of Prototypes for Distribution

This section evaluates the use of prototype-based languages in a distributed environment. The previous section has already addressed several issues in distributed computing in general. Here, the relevant differences between classes and prototypes will be highlighted in that context. These differences will illustrate the problems encountered with classes and will be an indication that prototypes with their simpler yet more flexible delegation scheme are better suited to structure large, dynamic and evolving distributed applications. This position is also advocated in (Dedecker and De Meuter, 2003; Dedecker et al., 2003).

- Object mobility is arguably a lot easier to accomplish in prototype-based

[6] This means that the network remains hidden to the user

[7] This means that the network is around us everywhere

languages than in their class-based counterparts. The main issue here is that the object's class will have to move along with the object. The reason is that an object simply *cannot* exist without its class. The class defines its behaviour and its interface. The situation is even aggravated because the object's class depends on its superclass, which in turn depends on its own superclass and so on. Moving an object therefore implies moving the *transitive closure* of its class as well! Each time an object visits a node that does not know about its class, the class (or classes) needs to be duplicated at the new node. This results in a time overhead because the class needs to be moved as well, and a space overhead, since a class definition is now stored twice.

Although objects in prototype-based languages "know their own methods", they will have to drag those methods along too. To ensure that an object's methods are co-located with the object itself, the methods should move when the object does. Thus, considering efficiency, prototypes will not always offer *cheaper* object movement but they will offer *simpler* object movement.

- A problem emerging from duplicating classes at different nodes is class inconsistency: if the original class is modified after it has been duplicated, these changes should be propagated to all nodes that have a copy of the class. Even for methods which are usually immutable in class-based languages, this can be problematic as updated versions of a class can be coded and recompiled. Class consistency becomes entirely problematic when considering static or class variables that are mutable: when replicating classes, the consistency of the class variable must be maintained. This necessitates a *replication management* system that can cope with the presence of concurrency often found in distributed systems.

- A related issue regarding class consistency is class version management, where the same class is updated and therefore possibly becomes inconsistent with objects instantiated from a previous version. During the execution of a distributed program a class will be replicated across different nodes. The versions of these classes will have to be kept consistent. If one of the nodes is equipped with an updated class, and an object of this class is moved from such a node to a node storing the old class version, what should be done with the object? Using an obsolete version of a class is not always possible as it could lead to erroneous interactions between state (the object) and behaviour (the class). The detection of "similar" or updated classes is not straightforward either. Class names may not be globally unique, and bitwise comparisons may not always give correct results.

- Embedding- or concatenation-based languages like Obliq are extremely well-suited to distributed applications (Cardelli, 1994). In Obliq, an object is entirely self-contained and has neither parent object nor class. Because of this self-contained nature of objects, Obliq readily lends itself towards network

applications. Cardelli (1994) argues that a delegation-based model of prototypes poses more problems, since it is more difficult to obtain complete relocation of an object and its methods. To a certain extent, this is true, and related to the difficulties of representing split objects. Delegation-based schemes have their own advantages, however:

1. They allow for better sharing, since objects can share their parents even if they move. Parent sharing thus allows for a very natural inter-node sharing and – equally important – communication mechanism, avoiding the intricate replication management schemes introduced by duplication of state. Moreover, even delegation-based prototypes allow for code to be local to a given object. For example, the language dSelf introduces *local methods* to minimize network traffic (see section 4.5.3).
2. Delegation-based languages like Self and dSelf can easily change their own behaviour by changing their parent slots. This is generally not possible in concatenation-based languages like Obliq. In dSelf, this is particularly useful as it introduces this flexibility of change in a distributed setting (Tolksdorf and Knubben, 2001). Objects can then themselves decide whether their parent (such as their traits) should remain remote, or whether a local object can take its place. Some precaution is needed here, since the safety and security of programs, extremely important in a distributed setup, are not taken into account in (Tolksdorf and Knubben, 2001).

Therefore, delegation-based prototype-based languages should not be ruled out a priori.

- Adding new *kinds* of objects (i.e. adding new data *types*) at run-time – without stopping the program – can be problematic in some class-based languages. In prototype-based languages, dynamically modifying or extending an object (even with new methods) is no problem. New kinds of data abstractions can easily be introduced at run-time. In Java, extension of classes at run-time is possible through the use of class loaders (Schougaard, 2003). Such extensions remain very restricted, however, due to Java's sandbox security model. In general, extensions through such abstractions as class loaders are certainly less expressive and much more tedious than simply extending objects from within a flexible prototype-based language.

- One may argue that using flexible, dynamically typed prototypes will deteriorate security. It is indeed impossible to find out what kinds of objects are accepted from remote devices and what they will be doing. There is no type or class to verify an object's behaviour. On the other hand, if the principle of Extreme Encapsulation is applied, objects can protect themselves. It should be the responsibility of the programmer to encapsulate his objects and to distrust any other object in the system. So, secure prototypes *are* possible.

Conversely, statically typed classes are themselves not a guarantee for safety either. As already mentioned in section 4.2.3, static typing can be partially circumvented by polymorphism. One can argue whether static typing is therefore necessary at all. When looking at a lot of contemporary Java code, one often finds code that bulks of downcasts and `instanceof` tests, very often because one loses all type information using methods that manipulate `Objects`. This nullifies all benefits of static typing. Thus, the static typelessness of prototypes can be certainly regarded as a blessing rather than a curse. Very often, static types interfere with distributed programs (because they are for example unknown by the receiving process). Also, in distributed systems making use of generated stubs and skeletons (like Java-RMI), static types often interfere with dynamic "proxy classes". Program code expects an object of a certain type T while the object being returned is actually of some type $TProxy$. To a certain extent, these problems are related to the ones outlined in section 4.2.5.1. These problems demonstrate that static typing and classes are not a *de facto* necessity in distributed systems. Objects can easily live without them.

- A fundamental property of parent sharing is that it is *explicit*. The programmer perceives the parent as being shared (possibly across the network) by multiple objects. It is thus his task to keep this parent consistent. This makes parent sharing a scalable sharing and communication mechanism, of which the programmer has total control. On the other hand, the automatic management of duplicated classes, class variables and class versions in class-based languages should be, but is not, transparent to the programmer and extremely hard to maintain for the language environment. In giving the programmer control of the sharing mechanisms, he can decide what properties it must adhere to. This will allow for more *transparent* programs, in which what is going on at runtime is visible in the source code, yet without having to deal with low-level concepts. Explicit parent sharing might give the rise to the impression that it would be a more low-level approach, since part of the responsibilities are delegated to the programmer. However, as will be demonstrated in chapters 5 and 6, the key to explicit parent sharing is providing just the right language features that allow for a convenient control over an object and its data by its children.

  Note that we do not advocate the use of parent sharing to support code reuse or code sharing across a network. Such a structure would lead to unacceptably high network traffic, since method lookup would involve network calls and increased network traffic. Rather, parent sharing should be used to support an expressive communication mechanism, in which changes appear to be automatically broadcast to all children. It should not be used to simply reuse parent methods in possibly remote children. If such a reuse *is* wanted, controlled mechanisms like Self's "copydown" can be used to avoid the need for manual duplication of code. We will explore such locality-promoting

mechanisms in section 6.5.3.

- Regarding performance, one cannot possibly argue that distributed communication using parent sharing would be of inferior performance than distributed communication occurring in class-based languages. After all, at the implementation level, remote objects become proxies that forward method calls. Whether these proxies point to a remote class-instantiated object or to a remote prototype-parent object is irrelevant to the communication at hand.

As can be witnessed from the above arguments, opting for prototypes is not a panacea that will suddenly eliminate all problems associated with classes in a distributed context. We have merely argued that they provide a more *flexible* alternative with less problems. They are conceptually simpler (for programmers as well as for language implementors) and better subject to change (able to cope with new object types at run-time in an *expressive* way). Breaking the dependency between object and class appears to be beneficial when one starts to move objects around. The dependency otherwise adds the burden of having to "drag along" classes, which can become inconsistent across virtual machines. Prototypes are more robust: methods are associated with objects and if updated prototypes are created, clones of an older prototype will not break down all of a sudden.

4.5 An Overview of Distributed Programming Languages

The discussion on the issues in distributed programming language design will now be grounded by reviewing some existing distributed programming languages. A small overview follows in which we motivate the choice for discussing each language.

Emerald (Hutchinson et al., 1991) is an object-based language with a strong emphasis on *object mobility*.

Argus (Liskov, 1988) addresses important concerns such as *transaction management* and *fault tolerance*.

dSelf (Tolksdorf and Knubben, 2001) is a true prototype-based language which features *distributed inheritance*.

Obliq (Cardelli, 1994) is a prototype-based language based on embedding centered around robust *self-sufficient objects*.

Borg (Van Belle et al., 2000) illustrates the concept of mobile interacting agents, supported by *strong mobility*.

4.5.1 Emerald

Emerald (Hutchinson et al., 1991) is a highly flexible and efficiently implemented distributed language. It is interesting in the context of our study because it is *object-based*: Emerald programs are inhabited by objects and by objects alone. The language was explicitly designed to simplify the construction of distributed programs (Jul et al., 1988). One important design concept that has added towards this simplification is the support of *fine-grained object mobility*: even the smallest possible objects can be moved to another node.

The other design dimension adding to the simplicity of developing distributed programs is Emerald's location-independent object invocation. That is, whenever a message to an object is sent, it does not matter *where* the receiving object is currently located. It is up to the Emerald runtime environment (the *kernel*) to locate the target of an invocation request (Jul et al., 1988). This simplifies distributed programs as the programmer does not have to distinguish between local and remote objects.

One of the reasons why the designers of Emerald have chosen an object-centred approach is that objects provide an excellent abstraction mechanism for building distributed applications because they provide the units of processing and distribution (Black et al., 1986). Objects are self-contained entities that provide excellent encapsulation barriers in a distributed environment. To give a general flavour of the language's look and feel, consider the following example that prints the squares from 1 to 10:

```
const Test ==
   object Test
      process
         var i : Integer
         i <- 1
         loop exit when (i>10)
            stdout.putint[i*i]
            stdout.putstring[" "]
         end loop
         stdout.close
      end process
   end Test
```

Emerald strives for language concepts similar to the ones we seek to explore, described in chapter 6. These concepts try to simplify the complex object management often left to the programmer, which is a large burden in distributed programs. What is also interesting is that Emerald features both private, local, passive objects and shared, remote, active objects (Hutchinson et al., 1991). This section will start with an overview of Emerald's language concepts. Next, we zoom in on the concepts specifically related to distribution support and partial failure handling. It is

not our intention to explain the language in detail. Rather, we give an overview of how distributed programs become more expressible in Emerald.

4.5.1.1 A Statically Typed Object-Based Language

Emerald is a strongly typed language. Moreover, it is *statically typed*. One important distinction with most statically typed languages is that Emerald's types are first-class. Types are modelled by objects and every object has a corresponding type. The language was designed to support explicit data abstraction: all typing is done at an abstract level, independent of the implementation. A type thus specifies an *interface* which can be used with different implementations. The relation between implementations and types is actually many-to-many: a single object can *conform* to many abstract types (interfaces), while an abstract type may be implemented by many different kinds of objects (Black et al., 1986).

Notice the difference between Emerald's type system and those of eg. Java or C++. In Emerald, there is a richer form of polymorphism since any object can be replaced by any other object, as long as it has a *consistent* type. Type S conforms to type T if S defines at least all operations defined on T [8]. There is a subtle difference between allowing subtyping based on type conformity and subtyping based on subclassing. Type conformity relates two objects through their interface, that is, two objects conform if they share part of their interface (their operations). However, classes related through inheritance also share their implementation: subclasses can override operations, yet they also inherit all instance variables (Black et al., 1986). Like Self, Emerald can avoid this enforced "implementation inheritance" by separating interface from implementation.

Emerald features *no classes*, which again shows that languages oriented towards flexible distributed programs try to avoid the concept of a class as much as possible. Emerald *does* support a syntactic `class` construct, which is mere syntactic sugar. Class declarations are automatically transformed to a generator object which can "instantiate objects" and a type object denoting the type of the generated objects. Emerald does not feature object inheritance, i.e. it does not have any notion of delegation. It does allow single inheritance through the syntactic class construct. However, such syntactic inheritance is only used at compile-time to reuse superclass methods in child objects. At run-time, objects resulting from inheritance do not keep any record of their inheritance relationships (Hutchinson et al., 1991).

Raj et al. (1991) regard the absence of code sharing as an advantage in a distributed setting, since objects are not dependent on other (repository-like) objects. Although Emerald objects do not depend on a *class*, they do depend on a *type*, which will have to be moved together with the object. Therefore, types do reintroduce some of the problems regarding classes sketched in section 4.4. Types remain

[8] Type conformity is explained in more detail in the Emerald Language Report (Hutchinson et al., 1991).

less problematic than classes, however, since they don't contain class variables or implementation specifications.

Objects in Emerald are highly self-contained. In fact, their internal data fields can only be manipulated through invocations. No external access to an object's data is permitted. Access to an object's data fields is mere syntactic sugar for accessor and mutator methods (Black et al., 1986). However, as will be noted later, Emerald does not adhere to the Extreme Encapsulation principle as we stated it in chapter 2: Emerald defines a multitude of operators that manipulate objects without their participation (i.e. by circumventing message passing).

4.5.1.2 Concurrency Via Active Objects

Emerald introduces concurrency by means of the notion of an "active object" by allowing a process (commonly called a thread) to be attached to an object. An object can implement a *process block* whose code will be executed autonomously by a new process until the end of the block is reached. Objects can be declared as being *monitors*, in which case they guarantee mutual exclusion of all of their operations. This means that those objects can only be used by processes "one at a time". Explicit synchronization can be accomplished using `wait` and `signal` operations, reminiscent of Java's `wait` and `notify` messages. These concepts were explained in-depth in section 3.2.2.2, as part of the Java model so we will not go into more detail here.

4.5.1.3 Object Mobility and Distribution Support

We have already mentioned that object location is kept as transparently as possible in Emerald. Yet, it is not *entirely* transparent: there are several language operations that allow the programmer to explicitly change the location of an object, or to query for an object's current location. A statement of the form `move exp1 to exp2` moves the object denoted by `exp1` to the location denoted by `exp2`. To be more precise, this statement is merely a *hint* to the language kernel to move the object. The kernel is not enforced to perform the move. Objects can be made immobile by fixing them at a given location. The `fix`, `unfix` and `refix` statements serve these purposes. Trying to move fixed objects results in failures.

Remote method invocation closely resembles RPC models. Emerald uses synchronous semantics (Black et al., 1986). Arguments are passed using call-by-object-reference, as is common in most object-oriented languages. When performing a remote method invocation, this implies that a *remote reference* to the object is passed. This has potential performance penalties, since every usage of this object by the called method will result in more network traffic (Jul et al., 1988). As mentioned earlier, Emerald provides two rather sophisticated (more high-level) parameter passing semantics, namely *call-by-move* and *call-by-visit*.

Call-by-move implies that the parameter is *moved* to the call site (that is, the node on which the target object resides). Call-by-visit is essentially the same, but

will move the parameter back to the call site when the method invocation returns. The advantages of call-by-move are that it is more expressive and that the parameter can be transported together with the call. The obvious disadvantage is that it increases the cost of the call, but also that it may create more remote references at the caller's site (Black et al., 1986).

When moving objects, the questions posed earlier regarding object mobility are raised anew: how much of the object graph has to be transmitted? Emerald solves this problem by allowing objects to be *attached* to other objects. Whenever an object moves, it also moves its attached objects. Attachment is transitive, so any attached objects to attached objects also move (Jul et al., 1988). It is a unidirectional bond however: if a is attached to b but not the other way around, then b will not move whenever a is moved. By default, objects are not attached, so nothing is moved together with the object.

Note that the use of statements such as `move`, `fix` and `attach` breach extreme encapsulation: they allow for an object to be moved or fixed without that object's cooperation. Of course, the object's state remains encapsulated and can still only be accessed through message passing.

4.5.1.4 Partial Failure Handling

Although method invocation is location transparent, the location of an object can still be retrieved by the programmer. There are two reasons for allowing this. One is performance, which is obvious in that one might want to store collaborating objects on the same node. The other is reliability and availability, two important factors in a distributed system.

To obtain reliable software, objects should not be stored on only one node, but they should be kept distributed by the system to keep the program resilient to crashes: whenever a node crashes, the program may still continue to work using objects on the remaining nodes. To facilitate this reliability, Emerald introduces a `checkpoint` statement, which allows an object to save all of its state to stable storage. This allows for object reconstruction in case of a node crash.

Emerald also supports exception or failure handling through the use of *failure handlers* (Hutchinson et al., 1991). These handlers can be compared with a "catch clause" commonly used in languages like Java and C++. However, since *any* call can generate failures (since it is possibly remote and the remote node may have crashed), it would become impractical to wrap each call in a so-called "try-catch" block. Handlers allow for the decoupling of the try-blocks and their corresponding catch failure handling statements. A special kind of failure is the unavailability of objects. To this end, Emerald allows for *unavailable handlers* which are used whenever an object is needed that can no longer be found. This is handy to deal with partial failures, yet the programmer has to explicitly provide such handlers.

4.5.1.5 Conclusion

Emerald is a high-level language, specifically designed to aid in the development of distributed programs. It introduces a single object model for small, local objects as well as for large mobile objects. Its type system is special in that types are first-class and allow for a total decoupling of interface and implementation. This relaxes the use of typing in a distributed setting, since it becomes less problematic to move types together with moving objects. Emerald adheres to the vision that distributed aspects should be transparent for method invocation, but that distribution cannot entirely be kept secret from the programmer. It is still his responsibility to structure his system in an efficient way, and to deal with inherent problems in distributed computing, such as partial failures.

4.5.2 Argus

Argus (Liskov, 1988), an extension of the CLU language, was designed to deal explicitly with long-lived distributed programs. In such programs, online data is maintained for long periods of time, and reliability is a major concern. Examples of such systems are file systems, mail systems and inventory control systems (Liskov, 1988). The language tries to ease the burden of partial failures on programmers. A distributed system is fundamentally different from a common sequential application. A sequential program is either running *or* it has crashed. Distributed programs may be partly running *and* partly have crashed. Thus, part of the program has to respond to such partial failures. Argus introduces *guardians* and *actions* to aid the programmer in structuring his distributed applications, both of which will be explained in more detail.

Argus is a statically typed language. To acquaint the reader with Argus' look and feel, consider the following example, which defines a procedure that will transfer an amount of money from an account on one node (a bank branch), to another, possibly remote node. The withdraw operation might signal that there are insufficient funds to withdraw, in which case the exception is propagated:

```
transfer = proc(from,to: account_nr, amnt: int)
           signals (insufficient_funds)
   f: branch := get_branch(from)
   t: branch := get_branch(to)
   f.withdraw(from, amnt)
      except when insufficient_funds:
         signal insufficient_funds
      end
   t.deposit(to, amnt)
end transfer
```

4.5.2.1 Guardians

Guardians are a special kind of object with the purpose of encapsulating a number of resources (Liskov, 1988). Its resources may only be manipulated through special procedures called *handlers*. A guardian itself may contain a number of active processes. Whenever a handler is called by another guardian, a new process is created to process the call. A guardian always resides at a single node, but may be moved. It may also create other guardians, which it may place at a given node. Finally, just like in Emerald, handler calls are location-independent, which means that one guardian may use another without knowing its exact location.

A Guardian's state is extremely important, since it may not be entirely lost upon a system crash. That is why a Guardian contains two kinds of objects: *stable* objects which will survive a crash (i.e. their data is written to stable storage) and *volatile* objects, which will not. Upon a system crash, all running processes and volatile objects are lost. A special recovery process will be spawned when the Guardian recovers. This process can be used by the programmer to restore as much of the volatile objects as possible from the surviving stable objects (Liskov, 1988).

4.5.2.2 Actions

Since there are multiple processes running within a Guardian, one needs to *synchronize* them. Synchronization and failure handling are handled by Argus' *actions*. An action can best be described as an *atomic transaction*, that is: a piece of code that runs as if no other processes are active and which is either entirely executed or not executed at all. To be more precise, actions are *serializable*[9], meaning that executing a number of actions concurrently (i.e. interleaved) will have the same meaning as executing them all sequentially in some order. Serializability allows for concurrency without the processes interfering with one another. Actions are also *total*: they either complete entirely (i.e. they can *commit*) or are guaranteed not to have any visible effect (they can *abort*) (Liskov, 1988). It is this property that allows for the graceful handling of failures, since partially finished computations will not leave the system in an inconsistent state, but will revert the system to the last known consistent point.

Actions are only created when dealing with *atomic objects*. Operations invoked on such objects will be synchronized and can be made undone using actions. Synchronization is implemented via locks. Every operation on an atomic object is classified as a reader or a writer, depending on whether the operation modifies the object. Argus follows the usual semantics in allowing multiple readers, but only one writer to manipulate the object at the same time. Recovery in case of failure is implemented using versions. Whenever an object is about to be locked, its state is tagged as a "base version". The operation then proceeds and performs modifications only on a *copy* of the receiver. If the action commits, the copy *becomes* the

[9] Not to be confused with serialization as a synonym for object marshaling.

base version. Otherwise, the copy is just discarded (Liskov, 1988).

Argus is unique in that it provides atomic actions as being built-in into the programming language. This allows for an expressive way to handle transaction management. Combined with Guardians, which provide encapsulation and recovery facilities, Argus is again an example of a language offering tools to deal with the complex problems that often arise within a distributed application.

4.5.3 dSelf

dSelf (Tolksdorf and Knubben, 2001) is a distributed extension to Self (explained in section 2.5.1). It is a language with concepts that closely resembles those of our own language under extension: Pic%. Both are delegation-based object-oriented languages. Thus, studying the concepts used in dSelf can give us some insight in the use of delegation in a distributed context.

Tolksdorf and Knubben (2002) argue why they prefer a prototype-based language such as Self over other (class-based) languages in a distributed setup. They note that the class hierarchy defines several dependencies between language concepts. First, a subclass depends on its superclass, usually via the inheritance relation. Second, any object depends on its class since its behaviour is defined there. These dependencies hamper the mobility of objects in a distributed program.

A direct consequence of having to move classes along with objects (as already noted in section 4.4) is the need for version control over classes, as is witnessed in Java-RMI (Tolksdorf and Knubben, 2002). In Self, behaviour objects can be factored out (traits) which do not distribute the code as in class-based languages. Because there is no duplication, there is no need to propagate changes to replicas. Keeping the code shared and centralized also has the advantage of changing an entire distributed object hierarchy in a single stroke (Tolksdorf and Knubben, 2001). Of course, the downside is increased communication overhead.

One can argue that this kind of sharing can easily be achieved in class-based languages by sharing the class over the network. In implementation terms, an object's "pointer to the class" would then be a remote reference. The major difference is that such class pointers are usually *implicit* in class-based languages (i.e. the pointer is usually not accessible or mutable). In Self and dSelf, a parent pointer to a traits object fulfills this task. This pointer is mutable, and thus objects can easily switch from a remote traits object to a local object whenever *they* see fit. The advantage over class-based languages is again one of increased flexibility.

4.5.3.1 Distributed Instantiation and Inheritance

The two main contributions of dSelf regarding the addition of distribution to Self are (Tolksdorf and Knubben, 2002):

Distributed Instantiation A traits object describing the behaviour of an object, and the object itself containing the representation do not have to be co-

located. Also, any clone of a prototypical object can reside anywhere, independent of the location of the prototype or its traits.

Distributed Inheritance An object and its parent do not have to be co-located either. This means that there is delegation over the network.

Both concepts lead to more flexibility in a distributed program (Tolksdorf and Knubben, 2002). Objects do not have to be co-located, which decreases communication overhead when an object is moved (that is, only that object needs to be moved, and not some class-like object as in eg. Java-RMI). Since the traits object remains shared on one node, changes are easily made. However, this flexibility comes at the price of decreased efficiency.

4.5.3.2 Local Methods

The two distribution properties mentioned above have the disadvantage that they raise the global communication overhead because messages may need to be looked up in remote parents. To this end, dSelf provides *local methods*. Local methods are ordinary methods, nested inside other methods. They are only accessible in the body of the method in which they are nested. Such methods minimize network traffic, since they can be transported along with the object. That way, local method invocation never leads to invocation across the network and allows at least some more control over the network traffic (Tolksdorf and Knubben, 2001).

4.5.3.3 Referring to Remote Objects

Distribution in dSelf implies that slots can refer to objects that are either located on the local or on some remote virtual machine. To be able to gain access the objects of another virtual machine, a special type of slot can be installed in a Self object. This slot can then point to the *lobby* of the remote VM. The Lobby, being the main access point for all reachable Self objects (see section 2.5.1.3) provides an access point for all objects residing on the remote VM.

As an example of distributed inheritance, lets extend the `Stack` example introduced in section 2.5.1.3. The first thing that needs to be done to create a reference to a remote VM. This can be done as follows (Tolksdorf and Knubben, 2001):

```
lobby _AddSlot: "remoteVM" ConnectedTo: "URL to remote VM"
```

This gives us a possibility to use the remotely defined `stackTraits` object. Note that distributed inheritance is gained by making the parent of the object point to a *remote* object.

```
localStackPrototype <- (|
  parent* = remoteVM stackTraits.
  stack = array clone.
  top = 0
|)
```

dSelf, like Self, distinguishes between ordinary and primitive objects. dSelf stretches the distinction further, since it attributes different semantics to these kinds of objects when referring to them across the network. In the case of referencing an ordinary object, a remote reference is created. Primitive objects (like integers and strings) get copied, so no references will be introduced for them (Tolksdorf and Knubben, 2001). This is reminiscent to our own approach, discussed in chapter 6.

4.5.3.4 Conclusion

It is noted in (Tolksdorf and Knubben, 2001) that dSelf introduces concurrency implicitly with the mere existence of multiple virtual machines, which do not even allow threads within one VM. dSelf thus needs to cope with concurrency and synchronization issues, as any distributed language. dSelf's support for this is quite rudimentary, providing only a simple locking mechanism. A more sophisticated concurrency model is expected in future versions.

As for the implementation, dSelf was implemented in Java, using Java-RMI for communication between dSelf objects. This resembles our experiment setup as we will also use Java as the host language of our interpreter. The dSelf implementation consists of a compiler, which compiles dSelf objects into Java objects. These Java objects are then manipulated by the dSelf Virtual Machine. The dSelf language shows that *it is plausible to extend a delegation-based language with distribution support*.

4.5.4 Obliq

Obliq (Cardelli, 1994) is a distributed language which we deem to be interesting as it is in a way an antipode of the language we developed, and yet in another way strongly related to it. Like our language, it is a distributed language which is *object-based*, it has no classes. Objects can be written just as simply as in Self by listing a set of slots between curly braces. However, unlike our offspring, Obliq is not a *delegation-based* language, since delegation between objects is missing. For the composition of objects, it uses an *embedding* strategy similar to the one that was introduced in section 2.4.4.2 when discussing non-delegating languages. Obliq does this by defining a *generalized clone* operator that can take more than one object as an argument. The result is then an embedding of all object arguments into one autonomous object, which has (copies of) the slots from all objects. The example below shows how to extend an object with extra fields.

```
let unidirectional =
    { x => 3,
      inc => meth(self,y) self.x := self.x+y; s end,
      next => meth(self) self.inc(1).x end };

let bidirectional =
```

```
clone(unidirectional,
     {dec => meth(self,y) self.x := self.x-y; s end,
      prev => meth(self) self.dec(1).x end });
```

Obliq features *creation-time value sharing*, which promotes the *autonomy* of objects. This approach avoids constructing a class-like hierarchy[10] and also ensures that operators applied to an object are necessarily local unless the programmer explicitly chooses otherwise. This facilitates efficient management of distributed objects, which is one of the main goals of Obliq.

4.5.4.1 Obliq Operators

Obliq specifies four additional basic operations, next to object creation, that can be used to express more complex operations such as for example atomic object movement (Cardelli, 1994). First, there is the operation of *selection* or *invocation*, which performs a lookup in an object, followed by either a selection or an invocation based on the contents of the slot. If the content is a variable, it is returned to the caller. If this caller resides on a remote node the value will be copied. If the slot is a method slot, the contained method will be executed. In a distributed setting this means that the method is executed locally and that the result is returned to the invoker across the network.

Next, an *updating* operator is introduced which – when applied on method slots – can be seen as a limited method overriding mechanism. The mechanism does not offer full-fledged overriding since the overridden method is no longer accessible. The operator also allows for methods to be replaced by other values and vice versa. The third operation, the generalized clone, was already introduced above.

Finally Obliq also has an *aliasing* operation. Aliasing allows for an object to automatically forward a message to another object. Note that this mechanism is truly a message forwarder and not a delegation mechanism, since aliasing features no late binding of self. Aliasing actually behaves like an indirect method invoker, with the difference that it also forwards update operations and that it works for both methods and values.

4.5.4.2 Security

In a distributed setting security is an important issue (see section 4.2.3). Obliq features operators which may be invoked on objects who can in principle not defend themselves against their effects. This is a dangerous situation in distributed settings where a malicious host may freely perform operations on all objects. To address this, two concepts are introduced. First of all operators are called *self-inflicted* if they are applied by one object to its own self[11]. This can in general only be de-

[10] which proves to be an obstacle to transparent and simple distribution as we have discussed in section 4.4

[11] Note that there is never confusion about self since Obliq does not feature delegation with late binding of self.

termined at run-time. Imagine the expression `self.q.x`, it is easy to verify that the selection `self.q` is self-inflicted as it is literally a selection on self. However, to see whether the selection of `x` is self-inflicted we must know that `q` returns self, which cannot be checked statically.

The concept of operators being self-inflicted is used to provide some security. Since Obliq features several operations which are implemented by statements rather than by messages sent to an object, avoiding a *breach of integrity* could be hard to accomplish. Especially the update operator is problematic, since it allows for update operations to take place outside of an object. Cloning and aliasing allow for similar uncontrolled access to an object's fields. Obliq tries to solve this deficiency by adding the possibility to declare an object to be *protected*. One can only perform selection and invocation on a protected object. This means that a protected object can have full control on whether it is being updated or not. Put briefly, if the object does not implement a method that performs an update on itself, it will be immutable by other objects. However, an "all or nothing" principle applies here, as an object is either protected or it is not. Using a message-based approach with respect to the Extreme Encapsulation principle explained in section 2.5.2.4, an object can specify whatever form of protection it wants by simply overriding the messages that should be rejected.

4.5.4.3 Introducing Concurrency

Concurrency is introduced in Obliq through explicit thread creation using a `fork` operator, which can be supplied with a procedure that will be evaluated in parallel with the caller. Results may be accessed by `join`ing a thread. Two forms of synchronization exist. The first one is used to prevent race conditions that may arise in a multi-threaded program. Obliq allows an object to specify that it is `serialized`, thereby associating a Mutex with that object. Operators must first acquire the lock on a serialized object before they can perform their normal operation. This type of system may easily provoke deadlock by common programming patterns such as for example recursive functions. Usually reentrant locks are used to solve this type of problems , but Cardelli (1994) argues that they are both too general and too restrictive to be useful. This is why in Obliq uses the *self-inflicted* concept to determine what operations should acquire the Mutex. Self-inflicted operators should never acquire a lock, as they are always the consequence of a previous external call, which awaits the completion of this call to be able to return.

Another level of synchronization is semantic or conditional synchronization, which has no relation to technical problems such as race conditions. Semantic synchronization is concerned with the meaning of the program. For example, when performing a `dequeue` operation on an empty shared buffer, the calling process (typically called a *consumer*) should wait. To express these concerns Obliq offers the *watch* statement: `watch c until guard`, where c is a *condition variable* and guard is a boolean constraint. If this constraint succeeds the procedure just

continues. If the condition fails the lock is released, allowing other operations to be invoked. These other operations may `signal` the condition variables that are watched. If such a change is observed the guard is reevaluated after the operation finishes. If the reevaluation succeeds the lock is acquired and the interrupted computation is continued.

4.5.4.4 Distributed Objects

Obliq is more than a concurrent language, its main goal is to provide a solid object-based model for distribution. To this end we need to establish what Cardelli (1994) calls communication channels. Such channels allow for the exchange of values. An initial channel is created using the mediation of a name server. On this name server a process A can publish one of its objects and then another process B can retrieve a reference to the published object. From then on both processes can directly exchange information though method invocation arguments and results.

In Obliq objects are really tied to the location on which they were created. When objects are passed as a parameter actually object references are being passed around. These object references are copied when supplied as arguments just like other basic values such as strings and numbers. The binding of objects to their site is, however, not a strong constraint as the user can clone an object to his site or pass around procedures that spawn new objects for him on other sites. Also, the way to move objects in Obliq is to consistently (i.e. in one atomic operation) clone the object on the remote site and then forward all operations of the local object to the remote object through aliasing.

Procedures that spawn objects on remote sites provide the main workload towards distributed programming in Obliq. They are sent to the so-called *execution engines* which evaluate them on their site. The procedures can thus be seen as some form of agents, yet they are mere *closures*. This means that they are lexically bound to the environment of definition: any free variables in the procedure are looked up at their site of definition. Thus, the use of free variables can imply distributed lookup. If an agent does not use free variables, it allows him to be completely independent, sharing no data with its original site. Conversely, it is sometimes useful to maintain a few free variables to allow communication with the originator. Furthermore the *network-wide scope* (Cardelli, 1994) also prevents the unpredictable effects of the dynamic scoping alternative where free variables of a procedure would be looked up in the context of the receiver. This would be a serious breach of the encapsulation of that site.

4.5.4.5 Conclusion

We have specified that Obliq is in a way the antipode of our own approach, which we will thoroughly explain in chapters 5 and 6. Obliq, like our own language, features an object-based approach. Yet, Obliq allows external operators to work on an object, without giving the object the choice to allow this type of interaction or

not. In the light of the security essential to distributed systems this is a bad idea, hence the possibility to *protect* and object. We will solve these issues by adhering to Extreme Encapsulation: the object itself decides which operations are allowed and which are prohibited.

Obliq keeps objects autonomous self-contained entities. The programmer can use an aliasing mechanism to "reroute" references to objects. However, the alias does *not* have late binding of self, to avoid that a call, which may have to travel over the network, comes back to the original receiver. These schemes are devised to make an object as robust as possible, and to limit the usage of network resources to perform a selection or an invocation. Our approach is the exact opposite, as we will devise a language featuring shared parents over the network, to explore the dynamism that such a distributed sharing can introduce in distributed applications. In this perspective split objects (and the possibility for `this` to refer to a remote object) are intentional since they allow expressing certain communication patterns in a more natural way (Dedecker and De Meuter, 2003).

4.5.5 Borg

Borg (Van Belle et al., 2000) is a research artifact, developed at the Programming Technology Lab of the *Vrije Universiteit Brussel* in the late nineties. It is an outgrowth of the Pico language (see section 2.5.3). Borg was particularly designed to support the notion of *mobile agents*, which are autonomous entities able to travel around and which can interact with one another. It was mainly used as a framework to study such concepts as Agents, Strong Mobility and synchronization primitives. Borg has lead to the study of two major issues (Van Belle and Fabry, 2001):

- How to develop and provide a robust mobile component architecture.
- How to write code in these kinds of systems.

Both are important in the context of our study on distributed language concepts. Particularly the first issue is helpful, as we will build our own distributed language in chapter 6. The design of an agent system like Borg is not trivial, as it should provide transparent interconnection between agents, route messages, migrate agents, serialize messages and schedule computations (Van Belle and D'Hondt, 2000). Borg's application domains can be situated in both large distributed application (such as e-commerce), as well as in true mobile computing, using handheld devices. When using such devices, application migration becomes necessary to minimize storage, as well as to deal with the lack of computing power. Programs could then be migrated to faster machines.

4.5.5.1 The CBorg Mobile Multi-Agent System

In Borg, agents run concurrently on one or more machines. They carry their own data space and computational state (i.e. their own runtime stack). Agents can be

moved between different machines (Van Belle and D'Hondt, 2000). Other features include (Van Belle and Fabry, 2001):

- Strong Mobility allowing agents to migrate without explicitly starting or stopping.
- an Agent Communication Layer supporting transparent message sends between agents. This also implies the need for automatic serialization, as noted in section 4.2.5.2.
- a Location-transparent distribution layer supporting the automatic routing of messages between different machines.
- Synchronization primitives allow agents to wait for one another.

A particularly sophisticated part of Borg is its distribution layer. This layer is necessary to address a problem left untouched so far. When objects can move freely in a distributed system, this inevitably opens up the problem of keeping track of their location. A virtual machine should know the location of a remote object when one of its local objects sends it a message. It becomes necessary to *route* messages between several virtual machines. In Borg, this problem is tackled by using special nodes, acting as both *name servers* and *routers*. These nodes are composed in a hierarchical manner. When agents migrate, the appropriate routing tables are updated as the object moves between nodes (Van Belle and Fabry, 2001).

4.5.5.2 Strong Mobility in Borg

We will briefly explain how Borg has achieved Strong Mobility. The details regarding this concept were already explained in section 4.3. In essence, Strong Mobility is achieved in Borg by migrating agents. An agent encapsulates a computational state (a runtime stack), which is transmitted with the agent upon migration. Thus, agent migration enables Strong Migration. Three actions are required to prepare, guide and complete this migration (Van Belle and Fabry, 2001):

- The agent's *complete* state needs to be encapsulated (serialized). This includes its state, its behaviour *and* its computational state.
- This capsule is sent across the network. This part poses less and less problems with the increased availability and reliability of communication networks (Van Belle and D'Hondt, 2000).
- The agent must be restored and re-actived in his new environment.

Strong mobility is a direct consequence of the transmission of the computational state of the agent. The fact that this computational state can be transmitted (or even grabbed!) follows directly from its first-class representation in the Borg

(or Pico) interpreter. Recall from section 2.5.3.2 that the Pico interpreter is able to represent the computation "that still has to be done" as a first-class entity because it is implemented with explicit continuations. Moreover, this computation is even first-class within Borg or Pico itself, but this is irrelevant for the purposes of strong mobility: the important thing is that one can grab and thus transmit the computation at the implementation-level.

It remains to be said how an agent's behaviour is migrated. In section 4.2.5.2 we mentioned the problem of specifying *how much* of the object graph to serialize. When an agent migrates, its data graph – including local objects – is transmitted (and thus serialized). The root environment is always cut off and replaced by the root environment of the new host. This achieves a decoupling from the agent and the underlying resources made available by the VM through the root environment (Van Belle and Fabry, 2001).

4.5.5.3 Agent Communication and Synchronization

Agents communicate through message passing. More specifically, message passing is *asynchronous*, so the sending agent does not need to wait for an answer. The call immediately returns `void`. This asynchronicity is deemed extremely important when using agent systems in Wide Area Networks, since the agent does not have to waste time waiting for results. To make two agents communicate in a request/response collaboration, the first agent can give a reference to himself as an argument, denoting "where to send the result". The other agent can then *call back* on the first using this parameter.

Sometimes, agents need to synchronize at a given point in time. This cannot be done using the asynchronicity introduced by message passing. Therefore, an extra synchronization primitive, called `sync`, was introduced. Its behaviour is most simply understood by an example, illustrated in Table 4.1 (Van Belle and Fabry, 2001).

Agent1	**Agent2**
`sync("Agent2",[a,10])`	...
waiting	...
waiting	`sync("Agent1",[20,b])`
continue with a=20	*continue* with b=10

Table 4.1: Illustration of the Borg `sync` primitive

Synchronization is accomplished using *unification* of a certain pattern (the second argument to `sync`). The pattern can be a table, strings, numbers, variables or a wildcard. While unifying the pattern, variables get bound, which result in two-way communication, as illustrated in the example. The first argument can be a named agent, a table of agents (thus, more than two agents can be simultaneously synchronized), or a wildcard. In case of a wildcard, the agent will only synchronize with any agent whose pattern unifies with its own pattern (Van Belle and Fabry,

2001). The synchronization primitive works even with agents that reside on different machines (i.e. it is location-independent). This results in a very expressive synchronization scheme between agents, provided that agents know each others name. This is reminiscent to the requirements of CSP's rendez-vous (Hoare, 1978).

There have been some thought experiments in extending Borg with new synchronization primitives (Verelst and Van Belle, 2000; Van Belle et al., 2001). One of the issues that is discussed is that Object-Oriented programs are unbalanced with respect to send and receive: sends can usually be scattered throughout the code, while a receive is only possible at the method level. An asynchronous (i.e. non-blocking) `recv` primitive is added that checks whether a certain message has arrived. Furthermore, this model using asynchronous message send, message receive and explicit sync is compared to other communication models such as Actors (Agha, 1986), the π-calculus (Milner, 1993) and CSP (Hoare, 1978). The main issue with these models is their inadequacy to express communication in large scale agent applications operating in a WAN.

4.5.5.4 Conclusions

Borg, being an extension of Pico supporting strong mobility, is an important precedent of our attempt to extend Pico with both a concurrency model, as well as a minimal yet flexible distribution model. In Borg, the focus lies on *agent mobility*. Our model will focus more on distributed inheritance relations.

4.5.6 Summary

The purpose of this section was to introduce a number of existing object-oriented distributed languages. This is relevant for a number of reasons. First, it shows how the issues introduced in section 4.2 are handled in different languages. Second, it shows that most languages have their own way in dealing with these issues, depending on what kind of applications they target. Argus, for example, stresses long-lived collaborations between objects (transactions) using rather heavyweight objects called Guardians. Borg on the other hand, uses smaller, more flexible agents that do not have any protection against system crashes. Also, languages like Emerald and Obliq stress object autonomy using *concatenation*-based schemes, while dSelf stresses *delegation* across network links.

The reader will have noticed that next to introducing the tools these languages offer for distribution we have also briefly recapitulated the notions of concurrency in the languages discussed: Emerald's monitors, Argus' atomic objects, dSelf's locks, Borg's `sync`, ... This way we have introduced additional concepts and solutions to problems with concurrent programming. These solutions will be synthesized in our own approach to concurrency in our prototype-based language (discussed in chapter 5).

4.6 Conclusion

In this chapter we have introduced several concerns that need to be addressed when writing distributed programs. These concerns range from pseudo-political observations which divide the distributed world in administrative domains, to very technical such as security regarding object transmission. Most of these problems boil down to the question whether we can *trust* an object. If the firewall of a certain domain can verify that a received object is secure, and in return the object has a similar guarantee that its host cannot do anything without the objects consent, there is no problem. A lot of attention was devoted to such safety and security issues. Extreme Encapsulation has been put forward as a good step towards such security guarantees, regarded from a language designer's point of view.

Trustworthiness, however, is not the only problem to be dealt with. We need to consider how to deal with distributed objects. This encompasses how we get a reference to such an object (section 4.2.4) and how we can invoke methods on it (section 4.2.5). Related to this concern is object mobility since one of the ways objects may move across a system is by parameter passing. These concerns will be brought back to light when we explain our model in the following chapters. Other issues that are equally important are left unaddressed, to narrow the field under investigation. Examples of such broad subdomains left unexplored are distributed garbage collection (section 4.2.9) and partial failure handling (section 4.2.8).

Subsequently another important issue in modern distributed programming languages – *strong mobility* – was introduced in section 4.3. Since we strive for simplicity and transparency wherever possible, we refuse to accept weak mobility which forces the programmer to foresee what a computation should do when it is about to be moved. Therefore we believe strong mobility to be a better choice as any running thread of computation can be automatically transported without any noticeable effect.

After introducing these issues as general problems and goals we have evaluated the choice of prototype-based languages in a distributed context in section 4.4. This led to the observation that prototype-based languages are more flexible compared to class-based schemes since they do not impose for example copying of classes across the network with the associated problems of keeping these copied classes consistent. At the same time we have introduced the benefits of delegation in a distributed inheritance scheme. Though good arguments exist for concatenation-based schemes, we argue in favour of delegation, since it allows a sharing that can never be done using concatenation. As we introduce our model, we will demonstrate that the distributed object inheritance we introduce serves to express communication patterns with a remarkable ease.

We subsequently grounded the discussion on distributed issues by a review of a careful selection of object-based distributed languages in section 4.5. None of these languages shares our view on distribution, however. Obliq and Emerald lack delegation and dSelf has a poor and very simple concurrency control and security mechanism. Our goal is to introduce delegation combined with synchronization

and encapsulation and to explore where it can help to expressively deal with concurrency and distribution issues. This exploration is the topic of the next chapter which introduces our delegation-based view on concurrency.

Chapter 5

cPico: a Concurrent Pic%

This chapter introduces cPico, a language providing a concurrency model for Pic%. A concurrency model is the combined set of features and concepts introduced into a programming language to support the management of concurrent programs. By management, we usually mean the creation, destruction, synchronization and structuring of concurrent computations. Taking into account the vision outlined in chapter 1, it might seem strange why we would want to introduce a concurrency model in Pic%, since our primary goal was to create a distributed language targeted to evolve towards a real language aimed at Ambient Intelligence.

We believe that by creating a concurrent language first, we narrow the gap between a sequential and a distributed language. This is because in many cases, distributed languages have to deal with concurrency issues. Thus, distribution often implies concurrency. To see why distributed programs are usually concurrent, note that we can represent each virtual machine as some sort of process or thread. Since objects collaborate across several VM's, there is the need for synchronizing these processes. Of course, the "process abstraction" for a VM can be encapsulated at the implementation level, yet somehow it will have to be reflected in the underlying language that there are multiple "computation engines" present at the same time.

In our context, our distributed programs are always implicitly concurrent programs. It therefore seems only natural to go from a sequential Pic% to a concurrent Pic% – cPico – first, without already taking on the burden of distributed problems. The concepts introduced to sustain concurrency can then be reused to support the concurrency implied by our distribution model. dSelf is an example of a distributed language that has seemingly skipped this step, since the language offers only very poor and minimal concurrency constructs. The approach taken in dSelf was to first add distribution to the language, while the concurrency model will have to be extended in the future (Tolksdorf and Knubben, 2001).

Note that it is *not* our intention to support concurrency for allowing the implementation of efficient parallel algorithms. Rather, support for concurrency is essential due to multiple interacting programs as witnessed in the vision of Ambient Intelligence. This is an important design choice which has enabled the use of

more “high level” concurrency concepts, often not considered when the language is meant to support fast data processing.

We begin our discussion of cPico by situating the proposed concurrency model in the design continuum delimited by the actors and threads extremes, introduced in section 3.2. Next, in section 5.2, we introduce some preliminary modifications we have made to cPico compared to the original Pic% model. Section 5.3 introduces the necessary language concepts that allow us to write and manage concurrent programs. These concepts are then further elaborated upon in sections 5.4, 5.5 and 5.6. The former presents the concept of parent sharing in a concurrent context, which is one of the compelling reasons to investigate prototype-based concurrent languages with delegation. Section 5.5 then presents a more grounded discussion on the repercussions of introducing active objects in object delegation relationships. In section 5.6 we will illustrates several ways to achieve conditional synchronization in our language. Section 5.7 provides a more technically founded argumentation for the differences introduced between Pic%’s and cPico’s object model. Before concluding the discussion of cPico in section 5.10 we discuss a number of issues on the implementation in section 5.8.

5.1 Situating The Model

It is interesting to situate our concurrency model in the “design space” of possible concurrency models. This clarifies what kind of model we have constructed for cPico. We start by sketching the edges of this “concurrency model space”. We can delimit the space by two “extreme” paradigms or models. On the one hand, there is the *functional* extreme, being the actor model conceived by Agha (1986). On the other hand, there is the *imperative* extreme, the thread model as found for example in the Java programming language (Gosling et al., 1996).

The actor model has been discussed in section 3.2.1. Recapitulating briefly, this model promotes a clean, functional programming style based entirely upon message passing. The pure model forbids sharing of state (i.e. no shared mutable data). An actor resembles an active object (as explained later), meaning that actors can process messages autonomously. Actors use asynchronous message passing and asynchronous message passing alone to communicate. The advantage of the actor model is that it has no problems with race conditions or similar concurrency problems as it is largely functional. The disadvantage is expressiveness: by continually having to specify customer actors, computations quickly become cumbersome to manage and hard to understand.

The other end of the spectrum is covered by threads, discussed in section 3.2.2. This is a rather low-level model, frequently used to structure operating system programs. Although the basic thread model is very simple to understand, it is quite error-prone when actually using it to write concurrent software. In thread models such concepts as locks, semaphores and critical sections are introduced to guard certain pieces of code or certain variables in the program against concurrent

access. Subtle bugs are introduced when the programmer forgets a lock or unlock operation, and programs are prone to deadlock.

There are of course concurrency models for languages that do not fit in either of these two categories. Most concurrent languages inheriting from the actor model, like ABCL (Yonezawa et al., 1986) and ACT-1 (Lieberman, 1987) have loosened the sharing policies and have introduced other types of message passing besides asynchronous message passing alone. On the other hand, there are languages inheriting from the thread model but who have introduced some specialized constructs in dealing with them. The language Obliq (Cardelli, 1994) discussed in section 4.5.4 is one such language. It uses explicit *forking* and *joining* of threads and condition variables to provide for synchronization.

cPico will build upon concepts from both concurrency models. More specifically, our active entities will be heavily based on actors. Message passing will also have an "actor-like flavour" yet we will introduce special constructs to incorporate return values as well. We relax the demands of actors that there is no sharing of state. However, when allowing shared variables one automatically introduces the problems mentioned in chapter 3. To control and deal with them, we introduce some well-known concepts from the thread model. More specifically, we will sometimes associate locks with objects so that they are guaranteed to be used by only one "thread" at a time.

5.1.1 A Case for Concurrent Models: Fibonacci

To clearly distinguish between the two models outlined above and our own, an informal *feeling* is created of how programming concurrent programs in our model is done. We do this by illustrating how a simple recursive fibonacci function can be written, which computes the fibonacci number for a given number n. A standard textbook recursive definition of this function in Scheme can be defined as follows:

```
(define (fib n)
   (if (< n 2)
       n
       (+ (fib (- n 1))
          (fib (- n 2)))))
```

A very straightforward and naive concurrent implementation of this function will just spawn two new processes at each level to handle each recursive call. Although this is really a contrived example, it does illustrate how concurrency is *created* or spawned in a language. Moreover, since the two subcomputations have to return a result to be combined in an addition, it also illustrates *synchronization* in the language.

Using The Actor Paradigm We start out by writing this example in an imaginary actor language that stays close to the original actor model with a Scheme-like

syntax. The code is adapted from an example written in an actor language implemented in Scheme by Theo D'Hondt for didactic purposes (D'Hondt, 2004).

```
(define-actor (fibonacci)
  (define-method fib n customer)
    (become fibonacci)
    (if (< n 2)
       (send customer (result n))
       (let ((newcust (fibcustomer customer)))
         (send self (fib (- n 1) newcust))
         (send self (fib (- n 2) newcust)))))

(define-actor (fibcustomer customer)
  (define-method (result num1)
    (become (fibcustomer2 customer num1))))

(define-actor (fibcustomer2 cust num1)
  (define-method (result num2)
    (send cust (result (+ num1 num2)))
    (become (fibcustomer2 cust num1))))
```

The example defines three actors, each having its own thread of computation. The first actor defines a `fib` method, the other two define a `result` method. The three primitives `create-actor`, `send` and `become` have semantics as explained in section 3.2.1. Recall that `become` states the behaviour that an actor will use to respond to *subsequent* messages. The `fib` method of the `fibonacci` actor starts out as expected, by testing the stop condition and sending n as the result to a certain "customer" actor if it is met. In the recursive case, the fibonacci actor will call its own method "recursively", yet this time with a new kind of customer. This new kind of customer is either a `fibcustomer` or a `fibcustomer2`. Whenever a `fibcustomer` receives a result it will *turn into* (i.e. become) a `fibcustomer2`, which has a reference to the result of one of the recursive calls. When the second value arrives some time later, the actor has enough information to carry out the addition and send the result back to the original customer of the `fib` method. `fibcustomer2` is also called a *join continuation* because it synchronizes the evaluation of different arguments (Agha, 1990).

In short, concurrency is created by sending messages to (other) actors because message passing is asynchronous and actors will compute messages autonomously. This is a high level approach and it is entirely implicit in the code. Synchronization happens by providing customer actors that will plainly consume the result "once it is computed": the processing actor will explicitly inform another actor (a customer) of the completion of an operation. Writing a program using such customers has the same complexity of writing programs in continuation-passing-style. This is the main reason why actor programs tend to become unreadable and hard to program.

Using the Thread paradigm To express the fibonacci function in a threaded concurrency model, we use the Java programming language, whose concurrency features were detailed in section 3.2.2.2. The solution is given below.

```
public class Fibonacci extends Thread {
   private int n;
   private int result;

   public Fibonacci(int n) {
      this.n = n;
      start(); // start a new thread
   }
   public void run() {
      if (n<2)
         result = n;
      else {
         Fibonacci f1 = new Fibonacci(n-1);
         Fibonacci f2 = new Fibonacci(n-2);
         result = f1.fib()+f2.fib();
      }
   }
   public int fib() {
     try {
       join(); // wait for this thread to die
       return result; // then read out the result variable
     } catch (InterruptedException e) { }
   }
}
```

The `Fibonacci` class is declared to be a subclass of `Thread` which will make the code of the `run` method run in a separate thread of control. Starting a thread is done in the constructor through a call to the `start` method. Synchronization happens by explicitly "waiting for the thread to die" using the `join` method defined on threads. Note that if this method were not provided in the `Thread` API, threads would have to be synchronized explicitly using `wait` and `notify` methods, as explained in section 3.2.2.2. Data is passed by sharing (reading and modifying) variables (the `result` variable in this case). In a thread model, the programmer must create, kill and join threads explicitly. He has complete control over these computational objects, but this comes at the cost of increased programming effort. Threads are synchronized using joins or locks. Although it *may* seem that the thread solution is more "easy to program" than the actor solution, the former is much more unsafe, while the actor program also has some implicit guarantees such as the absence of race conditions or similar problems. Threads do not scale up as well as actors when dealing with these synchronization issues.

Using cPico In the model we present, grafted on Pic%, the fibonacci function can be readily expressed as:

```
fib(n) :: {
  do() :: if(n<2,
             n,
             fib(n-1).do()+fib(n-2).do());
  activate()
}
```

This function actually defines the behaviour of what is called an active object. Such active objects behave very similar to actors, as they compute their own messages in their own thread of control. Message passing between such active objects is *also* asynchronous, but our messages *do* return a result. What exactly happens is that the new active object is equipped with a method that either returns its argument or invokes the same method again on two new active objects. Concurrency is created in this program by spawning (constructing) new active objects (by calling the `fib` function). By sending a message to these objects, new concurrent evaluations are started as the body of `do` calls `fib` again. Notice the complete *lack of synchronization code*. Synchronization is entirely transparent: the + operation will implicitly "wait" for the subcomputations to return a result.

This section was meant to situate our model in the realm of possible approaches to support concurrency in programming languages. By using the extremely simple fibonacci "expressivity benchmark", we were able to show how the two extreme paradigms – actors and threads – cope with the creation, destruction and synchronization of computing resources. The solution as specified in our model shows that we favour solutions that have minimal impact on the code itself, benefiting expressiveness. This section has only scratched the very surface of such issues as concurrency creation and synchronization. Subsequent sections will elaborate on this. It will also mainly deal with more complex constructs such as delegation and how they impose problems (and provide solutions) for a concurrency model.

5.2 The Pic% Model Reconsidered

The previous section has provided a basic feeling of how concurrent programs are written in cPico. In the next section, details on the concepts introduced to support concurrency will be discussed. For now, let us assume that cPico organizes programs as trees of objects, which can share parents. Two children of the same parent can be manipulated by two different active objects. In such case the parent is *shared* between multiple "threads", and thus may be subject to the race conditions that were introduced in section 3.3.1. To avoid this type of problems, the cPico object model is quite different from the Pic% object model we have discussed in 2.5.3.3. This section will therefore briefly introduce the major differences between

Pic%'s and cPico's object model. Technical details and clear motivations for this differentiation can be found in section 5.7.

Unlike Pic%, cPico makes a distinction between "call frames" and objects. A call frame is a (usually temporary) extension of an environment in which a function body is evaluated. This frame stores arguments to – and local variables of – the function. It has been explained in section 2.5.3.3 how such frames can be "captured" using the native `capture`. These call frames constitute exactly Pic%'s objects. cPico will distinguish between call frames and captured objects. The reasons in doing this are related to scope rules, detailed in section 5.7.2.

In cPico, an object will no longer delegate lookup for variables. That is, if lookup for some variable named x is initiated in an object, and x is not found, an error will be raised instead of delegating the lookup for x to the receiver's parent. This effectively makes all cPico variables *private* instead of *protected*. That is, whereas variables can be freely accessed in Pic% by children, they can *only* be accessed by its declarator in cPico. The difference between call frames and objects is that call frames delegate lookup for **both** constants and variables, whereas objects will **only** delegate constants. Section 5.7 describes the need for call frame delegation and how such visibility rules solve some problems related to locking.

The same section will also illustrate how these particular scope rules for variables make it impossible to further employ dynamic scope. That is why cPico *reintroduces static scope*. However, as noted in (D'Hondt and De Meuter, 2003) and explained in section 2.5.3.3, storing the lexical environment as a part of functions introduces problems when cloning objects. In order to cope with these issues a somewhat peculiar lookup mechanism was constructed, which will be discussed in section 5.8.2. In short, the most important differences with Pic% is that cPico's object variables are *private*, and that the language is subject to lexical or static scope.

5.3 Concurrency Concepts

This section will discuss what language features we have added to Pic% to make it a fully concurrent language. We will first introduce the new kinds of objects that inhabit a cPico program. Next, we have a look at how these objects communicate and synchronize with one another. Only when all necessary concurrency concepts have been introduced will parent sharing be discussed in section 5.4.

5.3.1 Active Objects

We have already mentioned active objects on several occasions throughout this text. They were mentioned in the context of ABCL when talking about concurrency, in the context of Emerald in chapter 4 and in section 5.1 when discussing the actor paradigm. An active object is a *unification* of the notion of a process or a thread and an object (Briot et al., 1998). This means that an active object is in control

of its own methods. It will execute those methods *itself*. This is in contrast to a *passive* object, whose code is executed by some external process, or in our case, some external active object.

5.3.1.1 Rationale

The reason for choosing this *integrative* approach to express computational resources is mainly inspired by the vision that one encounters two kinds of objects when modelling the world. The world is inhabited by passive objects like chairs, desks, walls, apples, trees etc... and by more active or autonomous "objects" like persons, animals, cars, computers etc ... Of course the distinction can become very vague: is a computer an autonomous object? Although it can be "active" on its own, one can say the same about a tree. The point is that we notice that there are some things in the world that we would like to model as autonomous objects, having their *own* thread of computation. This is one reason for choosing active objects from a modelling point of view. From a more language theoretical perspective, active objects are a good choice since they minimize the number of concepts a programmer has to deal with by integrating some in *one* unifying concept. Moreover, in a prototype-based language where everything can be regarded as an object, we have not introduced some process construct that is "above" objects, rather, active objects – although special – remain *just* objects viewed from a conceptual angle.

cPico's model diverges from the actor model as it is found in e.g. ACT1 because we allow these two types of objects to co-exist, namely both active and passive objects. Rather than to enforce making all entities active, we allow a combination of both. We found it odd having to model a complex number, for example, as an active object. Our attempt is to reuse as much as possible from the actor model, but to introduce additional concepts where they enhance the usability of the model by facilitating the design and development of concurrent programs from a software engineering point of view.

5.3.1.2 Active Objects in cPico

An active object is created by using `activate` instead of `capture`. For example:

```
counter(n) :: {
   incr() :: n := n+1;
   decr() :: n := n-1;
   activate()
}
```

defines an active counter object that will regulate the increment or decrement of the counter itself. `activate` implicitly performs a `capture`, so that it gets a reference to the "current dictionary". Next, it will transform this dictionary into an active object by associating a process and a message queue with it. The current

dictionary activated by `activate` is called the "behaviour" of the active object. Note that inside the method of an active object, `this()` will point to this *behaviour*. That is, `this()` denotes a *passive* object. We have added the native `activethis()` which always returns the "active object executing this native". The differences between `this().m()` and `activethis().m()` are not to be confused: the first is a simple synchronous method call, while the second is an asynchronous self-send, which will schedule a message request in the object's own request queue.

5.3.1.3 The Anatomy of an Active Object

When considering the internals of an active object, one can distinguish three constituents:

- The active object's **behaviour**, which is in a sense very similar to an actor's *script*. It is represented by a simple passive object. In fact, this object will always be *serialized* (see section 5.3.2).
- A lightweight **process**, denoting the computing power of the active object. In our implementation, this component maps naturally onto Java threads. This thread will continually wait for new messages to arrive and process them sequentially. It will be automatically stopped whenever the associated active object is garbage collected (see section 5.8.3).
- A **message queue** containing all messages sent to the object left unprocessed. This is an essential and important component of Active objects, since it allows for an active object to accept messages while it is processing.

Notice the complete analogy of this setup with the typical anatomy of an actor (Agha, 1986). One major difference with the actor paradigm lies in the fact that an active object's behaviour may be *shared* between multiple objects. That is, the behaviour is not just a private, encapsulated part of the active object. It is a plain object that may be accessible by other objects without the intervention of the active object skeleton. Second, in contrast to the actor model, cPico does not feature a *become* operation that allows the behaviour to be changed from one object to another. The behaviour *itself* can be changed however, for example through the use of imperative mixin methods. Third, cPico's active objects have the capability to *reply* to messages sent to them. That is, unlike typical messages sent to actors, our messages do carry some "reply destination" (see section 5.3.3).

5.3.2 Serialized Objects

When considering multiple threads of control (multiple running active objects in our model), *intra-object concurrency* can occur. This means that there can be multiple active objects simultaneously executing a method of a certain object. In some

cases, this imposes no problems. For example, there is no problem when two objects want to read out some variable at the same time by calling some getter function. Problems occur, of course, when dealing with the mutation of variables. In such cases, we would want method invocation on some objects to be *atomic*. A method invocation is atomic if, during the method evaluation, only one thread is actively using the object. Other threads that want to access the object by calling a method on it must wait. Our notion of serialized objects closely resembles Act1's "one-at-a-time" actors (Lieberman, 1987). Several authors have defended the use of properly serialized objects. Taura et al. (1994) argue this makes reasoning about the behaviour of methods substantially easier, since we do not have to take into account all possible interleavings of method invocations. Similar remarks are made in (Meyer, 1993).

Objects that find themselves protected against multiple active threads are also called *serialized* objects. In serialized objects, requests are processed one at a time, usually in their order of arrival (Briot et al., 1998). To ensure such behaviour for *active* objects, a message queue is used, which will sequentially batch all incoming message sends. Hence, active objects are always serialized. For *passive* objects which have no associated queue, locking is used[1]. A lock which is meant to mutually exclude methods is also called a *mutex*. This terminology and the concept of serialized objects actually stems from an operating system structuring concept called a *monitor* (Hoare, 1974). In Java, an object implicitly receives such a mutex whenever at least one of its methods is declared `synchronized`. Any synchronized method or statement block must first acquire the mutex before it can continue.

In cPico, a serialized object can be created using the `serialize` native. Much in the spirit of `capture`, `activate` and `mixin`, this native returns a serialized version of `capture()`. A serialized passive object is equipped with a lock. This lock has to be *acquired* by any active object that wants to invoke a method on it. This acquisition will take place *after* method lookup and *after* actual arguments to the method are evaluated. When the lock is acquired, the calling active object is the "lock holder". It can then proceed and execute the method body. Before doing so, the lock holder will also add the lock to its personal *lock list*. The raison d'être of this lock list will be explained below and in the following section.

A lock release is somewhat more difficult to program. Locks are released whenever a method returns. This means that internally, upon the return of serialized methods, the lock is released and removed from the lock holder's lock list. To support tail recursion, an implementation has to be careful that these lock releases do not make the runtime stack grow if a method on a serialized object is invoked tail-recursively. The lock list is necessary to maintain in the case of exceptions: when the method body raises an exception, the runtime stack will be subsequently ignored. It is therefore important that upon handling the exception by the virtual machine, all locks within the lock list of the owner are released. The elements of

[1] In the current implementation, cPico locks are implemented using Java's locking facilities.

the lock list represent all outstanding locks that were scheduled to be released on the runtime stack.

One important note is that only *method invocations* will be properly serialized. Accessing constants on the object is possible at any time, however. Thus, when performing `o.m()` on a serialized object, we will try to acquire the mutex. When performing `o.m`, we will not. In this case, m will just be looked up in the constant part of o and when found the value bound to it will be returned. The rationale here is that such plain slot access can never mutate an object. Moreover, a method invocation currently running in the object will not be able to influence the result of the evaluation of `o.m` since m is a *constant* binding, and thus immutable. One has to remain careful however, since the value returned by the lookup might be mutable. This means that concurrent access could take place. For example, a table t declared inside o as `t[10]::void` is mutable. That is: `t := x` is an illegal expression, but `t[i] := x` is not! This also means multiple active objects can execute `o.t[i] := x` at the same time, leading to possible race conditions. Of course, accessing and mutating data in this way is highly discouraged. Moreover, a value should only be declared as "constant" if it is really meant to be immutable. In short, the serialized object *can* stay in control of the mutable table by providing accessor and mutator methods which will be properly serialized.

We have not opted to make an object serialized by default, since this would impose too much synchronization overhead. Objects properly encapsulated within an active object do not require the overhead of locking, both in terms of time and space. We do acknowledge that a safer scheme would be the introduction of default serialized objects and a `unserialize` native, but this would require purely sequential programs to consider concurrency-related concepts.

As with any general mechanism providing locking, our mechanism is prone to *deadlock*: when multiple serialized objects send messages to one another in a circular fashion, every object will be waiting for the release of the lock held by some other object. The result is a classical circular wait that makes (part of) the program block. Section 5.4.3 gives some examples of deadlocks that remain in our model.

5.3.2.1 Reentrant Locking

It is important that the mutex lock of an object is *reentrant*. This means that a lock holder is always able to re-lock the lock it has already acquired. This is an important property, since otherwise, deadlock arises the moment a recursive computation is performed. As an example, consider:

```
makeSerializedObject() :: {
   fac(n) :: if(n=0,1,n*fac(n-1));
   serialize()
}
```

If an active object o was to call the function `fac` on such a serialized object without a reentrant mutex lock, then deadlock would immediately occur. o first locks the object, proceeds with the method body and performs a recursive call, which requires to lock the mutex again. Thus, o would wait for the mutex to be released by itself. Using reentrant locks, o can acquire the lock as many times as it wants when it is the owner. However, we require that the object releases the lock an equal number of times as it has acquired it. In the implementation, a simple counter is increased to represent the reentrant nesting level of the lock. Upon reentrant acquisition it is incremented, upon release decremented. Only when the counter is zero will the lock be truly released. If we would not adhere to such a strategy, then imagine the consequences in the example above when o would call `fac(x)`. o would lock the object, and call `fac` x times recursively. Then, upon exiting `fac(0)` it would release the mutex (since it returns from a method invocation) and other objects would roam freely inside the object *while o is still active inside it*! Using the counter strategy, the lock counter will be decremented x times. Only when returning from the call to `fac(x)` will the lock be released.

5.3.2.2 Locking and First-class Continuations

There is an important problem when dealing with locking in combination with first-class continuations. When using first-class continuations, a programmer can make a "snapshot" of the runtime stack at a particular moment in time. Such a run-time stack is comprised of *frames*, each incorporating specific evaluation code. Frames responsible for unlocking an object are called *release frames*. The programmer can restore the run-time stack to any "snapshot" he has made before. Using the `continue` native, the programmer can actually make a non-local jump in the program. Since it breaks with the rigid call/return structure of ordinary function or method invocations, it also breaks with our simple locking strategy. First of all, if the programmer replaces the runtime stack by another one, he can discard release frames such that the virtual machine would be unable to release outstanding locks. Second, when jumping back to a certain saved continuation, it is possible that release frames are executed twice, meaning that a lock is released multiple times, resulting in erroneous behaviour. A program as simple as the following would then crash:

```
object() :: {
   m() :: call(cont);
   serialize()
};

c: object().m();
continue(c, void);
```

When the run-time stack is captured inside `m`, the frame to release the object's mutex will still be on the stack. When later on jumping back to this code, the re-

lease frame will be executed for the second time, leading to the release of an open lock. Because one is able to store a snapshot of the running computation inside a method of a serialized object, one can also break the atomicity guarantees on an object: when multiple active objects use continue in parallel with the same continuation, they can be active simultaneously inside of a method of a serialized object. Allowing only the active object that has *captured* the continuation to continue on it does not rule out this problem: an object can continue on its own captured continuation that contains a method invocation of m while some other active object is also executing m. The problem lies with the fact that locking inherently uses the normal call/return control flow to organize locking. This control flow no longer holds in a world with first-class continuations. Therefore, we will need to extend environments to explicitly deal with lock management.

Our proposed solution is to cleverly reuse the lock list of an active object. At the moment the runtime stack is captured, we can store (a copy of) the entire lock list of the active object in the captured environment. Recall that the lock list contains *all* locks currently owned by that object and for which proper release frames are on the stack. Whenever this environment is restored (i.e. `continue` is used to jump back to the encapsulated continuation), the caller of `continue` releases all of his locks. This is necessary to prevent currently locked objects from being locked forever. Subsequently, the stored lock list is retrieved and all contained locks are re-acquired before evaluation in the new environment is continued.

This scheme is implemented in our proof of concept implementation. Care had to be taken to ensure that reentrant locks were *completely* released when performing a `continue`. Also, if the captured stack contains a lock that is held reentrantly, the lock has to be re-acquired as many times as it was taken before the `call` upon continuation. One downside of the scheme is that if all locks in the lock list are not taken atomically, chances for deadlock to occur will increase. Deadlock could occur if some other active object is trying to acquire similar locks in the lock list at the same time.

5.3.3 Asynchronicity and Promises

Active objects, just like actors, communicate using “message passing”. In a sequential object-oriented language, message passing is synchronous: the sending object waits for the receiving object to compute the value. In fact, the sender does not really “wait” for the result to be computed, instead, there is only one active thread that transfers control from the caller to the callee and then back. In this section, we will discuss how cPico defines message passing semantics and how promises can be used to retrieve return values.

5.3.3.1 Asynchronous Message Passing

In cPico, message passing between active objects is *asynchronous*. This means that the sending object does *not* wait for a result to return. Rather, it will compute

in parallel with the callee that processes the message. This way, concurrency is introduced in programs. When dealing with pure asynchronous message passing, the message send returns no value. To be able to acquire a result, one has to work with explicit *callbacks*. A small example will illustrate this:

```
caller() :: {
   m(callee) :: callee.n(caller);
   handleResult(result) :: display(result);
   capture()
}
```

In the example, an object `caller` will perform an asynchronous method call to a parameter `callee` object. To be able to acquire a result, the caller passes itself as a parameter to the method. The callee will then "call back" on the caller through the `handleResult` method. This kind of code often gets quickly unreadable as it pulls apart the context of where the call is made and where the result of some computation is used further on. Moreover, we need some mechanism to link the "call site" with the "result site", that is, we need to make sure which result belongs to which method call. To alleviate such problems, the concept of a *promise* is introduced.

5.3.3.2 Promises

The concept of a promise or a future has already been explained in section 3.4.2 when discussing ABCL's "future type message send". Put briefly, a promise is a "placeholder" for the result of an asynchronous method invocation. That is, the result of the method call will be stored inside this placeholder. Each method invocation on an active object can be thought of as having such an implicit promise, which will always be fulfilled (implicitly) by the value of the last expression in the method body. Whenever an active object tries to *use* the placeholder when it has no value yet ("when the promise has not yet been fulfilled") it is blocked and waits for the value to arrive. If someone performs an operation on an already fulfilled promise, the promise forwards the operation to its underlying value and thereby becomes invisible to the programmer. The example above can be rewritten using promises as follows:

```
caller() :: {
   m(callee) :: display(callee.n());
   capture()
}
```

The method call to n returns a promise, which is subsequently accessed by the `display` native. This will make the caller block until the method has been computed.

Promises or futures are no new concept and have been used in other concurrent and even distributed languages in the past, primarily in the late eighties. Example languages are Multilisp (Halstead, Jr., 1985), ACT1 (Lieberman, 1987), Argus (Liskov and Shrira, 1988) and Eiffel// (Caromel, 1990). In Multilisp, futures are used to start up concurrent computations. Futures act as placeholders for the result of the computation. Accessing them before they are determined leads to an implicit *join* synchronization. In Argus, promises are not transparent. Rather, they are plainly used to get a handle to the return value of an asynchronous method invocation. In Eiffel//, promises are called *awaited objects*. Their key role there is to support *wait-by-necessity* (Caromel, 1989), whose main benefit is that it allows for a smooth reuse of sequential programs. These programs can be parallelized without having to add synchronization code. The implementation of cPico promises in terms of Java synchronization primitives will be discussed in section 5.8.1.

One important reason for using promises is that they minimize the cognitive load on the programmer when he has to deal with asynchronicity. This cognitive load is much higher when forced to use callbacks or customers. Ideally, promises are fully *transparent*, meaning a programmer *cannot* distinguish between a promise and its fulfilled value (in type, interface, behaviour or even identity). This is the case in cPico. It is also a major distinction between cPico promises and ABCL futures, which are not transparent. Also, unlike ABCL's futures, cPico promises can only *become* one value, they are no queues. Fully transparent futures are highly wanted in languages employing them, as can be seen from several attempts in adding futures to popular languages like Java (Pratikakis et al., 2003) and C++ (Chatterjee, 1989). These approaches of *adding* futures to an existing language are usually flawed because they interact in unpredictable ways with existing language features, like equality testing, downcasting, type systems etc... Since we have built a cPico interpreter ourselves, consistently integrating futures as part of the language has been substantially easier (see section 5.8.1). Also, as mentioned in section 4.2.5.1, the cPico language itself can more easily cope with futures as it employs dynamic typing and uses message passing instead of operators.

5.3.3.3 Message Ordering

One important question concerning asynchronous message passing between active objects is the execution order of received messages. For example, given the following code, where `stack` is active:

```
{ x: stack.pop();
  stack.push(5);
  y: stack.pop()
}
```

How are the `push` and `pop` messages on this stack going to be executed relative to one another? Even though the messages are sent asynchronously, we would

expect them to be executed in the order they were sent. In our model, this is the case, but it does not necessarily have to be. Our model follows that of ABCL/1. In (Yonezawa et al., 1986) this principle is stated as the *Assumption for Preservation of Transmission Ordering*:

> *When two messages are sent to an object T by the same object O, the temporal ordering of the two message transmissions (according to O's clock) must be preserved in the temporal ordering of the two message arrivals (according to T's clock).*

Yonezawa et al. (1986) also note that this assumption was not made in the actor model, which can therefore sometimes be hard to understand. At first sight it may seem that this ordering is trivial to implement when messages are sequentially stored in a message queue, since a queue imposes the ideal linear ordering we would like. Things become more complicated when *guards* are introduced to delay messages to be executed. When a guard fails, we cannot just perform round robin scheduling, as this would lead us to busy waiting. Instead, messages that cannot currently be processed because a method is unavailable must be put in a separate queue. If we then want the above transmission ordering property to hold, we have to be careful not to execute any message sent by an object which has some messages pending in the "unavailable queue".

5.3.4 A Uniform Active Object Model

We have introduced active objects as objects having the ability to evaluate code autonomously. One important aspect of an implementation is to try and keep the interpreter as uniform as possible. Therefore, the "top-level processing power" which will evaluate the input typed in by the programmer (and therefore will evaluate *source text*), is also modelled as an active object. This object is called the `main` active object. `main` is a plain active object whose behaviour is defined as the `root` dictionary. Whenever the user evaluates some program text, this is defined as an asynchronous method invocation of `eval` on `main`, sent by `main` itself. This method invocation – like any other active object invocation – immediately returns a promise. The interpreter will subsequently try to print the "result of evaluation", which is this promise. Thus, it will automatically wait for the result of evaluation (i.e. for the promise fulfillment).

Evaluating `activethis()` at top-level results in a reference to the `main` active object. Evaluating `this()` will evaluate to the `root` dictionary. Note that `root`, like any other active object behaviour, is serialized. This might sometimes lead to subtle deadlocks since it means that *all* top-level functions are automatically declared mutually exclusive. If this is not wanted, the programmer has to create an unserialized view on the root and declare his functions in this view. Although we know this is sometimes very clumsy for the programmer, the uniformity of the model should not be broken since we want calls to `activethis()` to be atomic.

It is also this property that allows the user to perform multiple evaluations without waiting for the previous evaluation to finish.

5.3.5 Concept Overview

The most important concepts introduced in cPico will now be reviewed briefly. The addition of concurrency in the language is introduced by *active objects*, very reminiscent to actors. Messages sent to active objects are processed *asynchronously* and their corresponding methods are executed by the receiving active object itself. The sender of such messages receives a *promise*, a placeholder promised to be fulfilled with the return value of the method call. Accessing the promise's value ("touching" a promise) before it is fulfilled synchronizes the accessor until the value is available.

Whereas active objects are always *serialized*, passive objects are not. Passive objects can be explicitly turned into serialized objects using the `serialize` native. Any method call on this object is guaranteed to be atomic. In what follows, the structural relationship of objects through delegation is studied in the context of concurrency and the newly defined cPico language concepts.

5.4 Parent Sharing

In our model we will strongly advocate parent sharing as a solution to a variety of problems. We have already explained the term in our introduction on prototype-based languages. Parent sharing is one of the key features of prototype-based languages with delegation. Since it is possible to create two extensions from the same object these two objects both have a sharing relation towards the same parent. As such the children can use this common parent to communicate with each other. We call this parent sharing. In this section we will demonstrate that using this type of communication can have a variety of advantages, although they are also due to another powerful feature of Pic%, called encapsulated mixin-based inheritance. Using this type of inheritance which we have explained as part of the Agora model in section 2.5.2, a parent can keep control over the objects that become its children. This allows a form of control which allows us to characterize parent sharing as a relation with many desired properties in a concurrent context.

5.4.1 Scope Functions

We have already mentioned that in our revised model children are not allowed to see variables of their parent anymore. As such they are subject to the same restrictions as any other object since they can only use the public interface of an object. However, when we model certain objects to be children of some parent this relationship should imply that the object has some additional privileges. One of these rights that is always respected is function overriding. If the parent object

uses `this().m()` in its code then child objects may override this function with specialized behaviour.

Additional privileges for children are offered through what we call *scope functions*. The first of these scope functions is a generalization of the `super` native which takes a single expression as argument. This expression is then evaluated in the context of the *parent* object. This allows for access to the variables of the object's parent. Consider the following simple example, which could be used to calculate the sum of three numbers which can be supplied at different times.

```
parent(x) :: {
   getx()::x;
   child(y) :: {
      m(z) :: super(x) + y + z;
      capture()
   };
   capture() }
```

Notice the `child` does not have to use the function `getx` to access x, since it can have direct access by evaluating `super(x)`. In section 5.2 we have stated that cPico splits the concepts of call frames and objects to prevent objects from delegating lookup for variables. This is due to the fact that delegating lookup for variables can cause a plethora of locking problems, as will be illustrated in section 5.7. However the `super` scope function does not compromise the model, since it can be seen as the ad-hoc definition of a function `lambda()::exp` which is then immediately called. That is why we can regard the argument to the function as being a *critical section*: it is guaranteed to execute when no other active object is using the parent object. Of course a lock is only acquired if the parental object is either serialized or active.

A similar native exists that allows to revert to the scope of the receiver, possibly from within a `super` invocation. Because super sends adhere to late binding of self, the following relation holds: `this() = super(this())`. In other words, when using `this()` in a super scope function, we can access the initial receiver. `this` can also be used as a scope function, of which the single argument expression will be evaluated in the receiver. We could for example rewrite the method `m` from the example as `m(z) :: super(x + this(y)) + z`. Figure 5.1 shows another example of such an interaction. We have a general chat client which has a chat-buffer and a specialized child which features some graphical interface window. In the update methods we wish to update the buffer with the contents of our window, we use `super` to access the chat-buffer variable. To get the contents of the window we need to go back down using the `this` scope function.
Notice that `this`, like `super`, tries to acquire a lock on a serialized `this()` first, ensuring that race conditions cannot occur. We refer to appendix A.2.1 for a more formal semantics of scope functions.

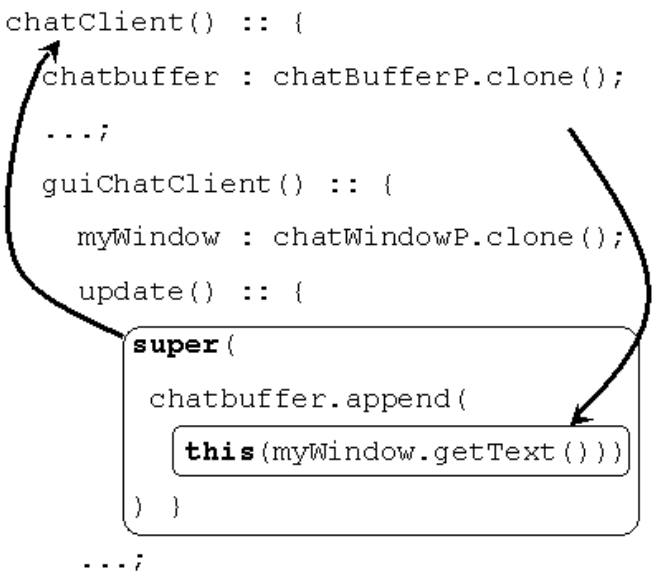

Figure 5.1: Example of using Scope functions

The idea of evaluating some code issued by a parent in one of its children (in our case using `this(exp)`) is not new. The programming language Beta (Lehrmann Madsen et al., 1993) offers an `inner` construct, enabling a superclass to implicitly use code of one of its subclasses. This construct can be used to directly express *template methods* (Gamma et al., 1995). For such methods, it is natural for the parent or superclass to stay in control of the global pattern. Parts of the pattern which it cannot directly solve itself are then delegated to a child[2].

5.4.1.1 Cloning Revisited

There is a general problem when combining cloning with extreme encapsulation. The problem is that when an object creates a clone of itself, it finds the clone already encapsulated and hence cannot initialize it with other values for instance variables unless the clone provides a public interface for doing this. However, defining a public interface to mutate one's instance variables would nullify all benefits of extreme encapsulation, because one then has getter and setter methods for instance variables. For this reason we introduced a `cloning` native that takes an expression as argument, which causes the expression to be evaluated in the context of a freshly allocated clone of `this()`. The return value of `cloning` is the initialized clone. For example, cloning a cartesian point with x and y coordinates and initializing it to the origin would then be possible by evaluating `cloning(x := 0; y := 0)`.

[2]The resemblance between our `this` scope function and the Beta `inner` construct was communicated to us by W. De Meuter

Nevertheless, two problems remain with this approach to cloning. The first one is a problem that can also be observed with `super` and `this`, though it is more apparent with `cloning`. In the previous section we have already noted that scoping functions can be thought of as being expanded into nameless thunks. Because these functions have no arguments, they cannot be parameterized with useful values by the caller. Consider the example of the cartesian point, we cannot simply define:

```
clonePoint(newX,newY) :: cloning({x:=newX; y:=newY})
```

The reason is that the parameters of `clonePoint` cannot be used in the cloning expression. Our current implementation avoids this problem by evaluating the expression to `cloning` in a copy of the frame in which `cloning` is called. This frame will be placed under the clone of `this()`. This allows the programmer to use local variables in the call frame, as well as variables in the clone. The above example then becomes valid.

A cleaner approach with regard to scoping would consist of defining special *types* of methods. For example, we could define a special type of "cloning method", annotated through a different syntax. Such methods will then always be executed in the context of a clone. This alleviates aforementioned problems as the method can take parameters just as any normal method. The downside of introducing such methods is that it would break with Pic%'s extremely simple model where there is not even a difference between functions and methods. Section 6.1.3.3 will revisit these ideas.

A second problem with objects sharing their data with parents is whether or not the parent should be cloned if the child is cloned. In standard Pic% this was taken care of by a `clone` native which could be parameterized by a dictionary that indicated the last parent to clone. This native is still present in cPico since the `cloning` native is not sufficient to replace it. `cloning` can be thought of as using `this().clone(this())`, so the clone's parents would be shared by default. This is definitely not always wanted. Consider a cartesian $2D$ point having x and y coordinates, and a view extending the point to a $3D$ point having a z coordinate. When cloning the $3D$ extension, it seems logical to clone the parental $2D$ object.

5.4.2 Advantages of Parent Sharing

Parent Sharing has some advantages in a concurrent and distributed setting. First of all it provides extra encapsulation which promotes security and safety. If all sharing is done through parent sharing we can use the scope functions we have just mentioned to gain access to variables, without providing accessors. If this technique is used consistently no hostile object can abuse any public accessor interface to gain access to variables. This enhances the *security* of the program, which is especially important in a distributed context, where programs or parts of it may be moved to harmful hosts. Furthermore this encapsulation also introduces *safety*. This means

that if the internals of an object are inadvertently changed, we should only look at the children of that object, instead of having to search the entire program for uses of the public interface. Thus, scope functions promote the basic principles of locality and modularity.

One additional advantage is that the shared parent is an ideal synchronization medium for several children that can be activated by different active objects. Possibly the children themselves even *are* active objects. They can communicate with the parent which is for this purpose of course at least a serialized object. This way they can periodically exchange information and the parent can serve as a safe communication channel. By having to use scope functions that are also critical sections, we can guarantee synchronization.

5.4.3 Deadlocks Using Parent Sharing

When using two objects that are either serialized or active (to avoid intra-object concurrency), deadlocks can easily occur if two such objects collaborate with one another in a composition relation. Although parent sharing can be used to provide more secure collaborations, deadlock can still occur between a parent and a child. First, consider the situation where deadlocks can occur using composition. An example is shown below where two serialized objects concurrently query for each other's name. Since both A and B are serialized objects, their mutex will be locked prior to method execution. Neither object will be able to continue since both require the lock on each other.

```
A :: {                          B :: {
  m() :: {                        n() :: {
    ...                             ...
    B.name();                       A.name();
    ...}                            ...}
  name() :: "A";                  name() :: "B";
  serialize();                    serialize();
}                               }
```

On several occasions we have already advocated the use of parent sharing as a safer way to write concurrent programs. However, even with programs that use only parent sharing, deadlocks may occur as the following example shows. A parent object initiates a dynamic lookup for a method n. This method can be seen as an abstract method of the parent. It knows that its children will implement it.

```
parent() :: {
   m() :: this().n();
   p() :: void;
   child() :: {
```

```
      n() :: .p();
      serialize()
   };
   serialize()
}
o: parent();
c: o.child();
```

However, this program can result in deadlock. Figure 5.2 illustrates such situations. Consider two threads A and B. A calls m whereas B will call n directly. As such, A has a lock on the parent object o and B has a lock on the child c. For either one to proceed a lock must be released. A solution to this problem is making the child an active object, where messages get serialized and will be executed one by one.

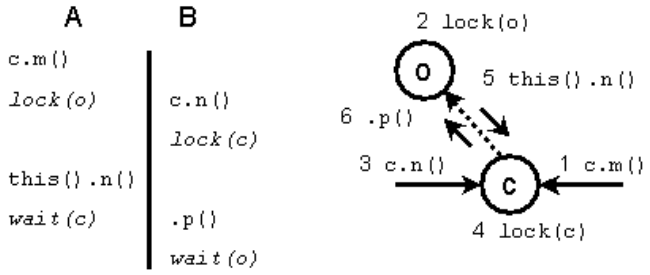

Figure 5.2: Deadlock between objects in a parent-child relationship

5.5 Mixed-Object Delegation Patterns

This section introduces some of the technical problems that are associated with the introduction of active objects as a separate entity in an object-oriented language. It will be explained how active objects can interact with passive objects through a delegation relation.

In order to model concurrent applications both active and passive objects are integrated into the model. Our intention is to stay true to an integrative approach, with attention to the usability of the language. The ease of programming is one of the main reasons to introduce both passive objects and the possibility of shared data. However, having to distinguish between two types of objects has some repercussions. If it would be possible to mix active and passive objects in a delegation chain, proper semantics for such relations must be defined. We will first consider all possible relations before proposing our desired solution.

Passive-Passive Delegation This is the classic case where we have a passive delegation chain. Here we have a traditional extension in mind, such as extending a $2D$ point to a $3D$ point. Naturally method invocation as well as delegation implies synchronous evaluation.

Active-Passive Delegation Here an active object is put underneath a passive hierarchy. This can be done to encapsulate a passive object that will be shared through parameter passing to some other active object. The queue of the child will ensure that only one message is computed in the hierarchy, to avoid for example the deadlock situation we have mentioned in the previous section. Again this system behaves as expected, method invocation on the child is asynchronous, invocation on the parent is synchronous. Moreover, super sends from child to parent are also synchronous.

Passive-Active Delegation A passive view is created underneath an active object. One way to envision the usefulness of such a hierarchy would be to see this passive view as a part of a split object, where the active object is the "identity holder". Imagine a `person` object, which is represented by an active object. The roles this person could play (a sportsman, an employee, ...) can be modelled as passive views. The question we need to ask ourselves then is whether we want method invocation on these roles to be synchronous. I.e. when someone calls `person.asSportsman().sport()`, do we want this invocation to be synchronous or not? In other words: is the property of being an active object inheritable?

- It might seem logical to request that the invocation is asynchronous and that messages are in fact scheduled, since they are actually operations on an active identity. However, this would imply that a passive view is implicitly active if it is a child of an active object. This is in fact confusing, in such a case it is better to use active views.
- When considering method invocation to be synchronous we run into the problem of defining semantics for delegation to an active parent. Since this parent is active it seems a reasonable request to demand that all messages sent to it, even through delegation, will be scheduled in the queue. This also gives us an awkward situation where delegation actually schedules a message, and thus if the method is not overridden, method invocation would suddenly become asynchronous even when sent to a passive object.

As the reader can see the clear semantics we deemed necessary cannot be provided in this case. Moreover we will observe a similar problem occurs when extending active objects with active objects.

Active-Active Delegation There are two ways to look at delegation from active objects to active objects. The first one is related to the role modelling we have discussed in passive-active delegation. Only now the roles are independent active objects. As such asynchronous message passing is acceptable.

- One way to deal with this technique is to allow that active objects share the same message queue. This makes sense for the person-sportsman-employee example, since operations on the person as an employee are

actually operations on the person itself. This introduces some problems of its own. First of all we need to store the object on which the method was invoked for every request, and secondly sharing of active object queues is *not* always desirable. For example, since the root object is an active object that is used to evaluate top-level code, all active objects (being necessarily children of `root`) would end up sharing one queue. This solution rules out any concurrency.

- The other approach is to create new active objects, with separate threads and message queues. However when a message needs to be delegated, we do not want to queue it in the parent as well.

The latter two cases demonstrate that the semantics of extending active objects are somewhat vague. Therefore we will make it impossible to make such extensions, not by forcing all active objects to be childless, but by separating the active object's **behaviour**, which is a serialized object, and its **active part**. The active part can be seen as the thread and a message queue, as explained before. This allows us to specify that any extension of an active object is in fact an extension of the *behaviour* of the active object. This way we only have the first two cases we have mentioned, and we obtain clarity, since we can stipulate the following rules.

1. A message sent to a passive object is always synchronous.
2. A message sent to an active object is always asynchronous.
3. Messages delegated by passive objects are never subject to scheduling. Supersends or self-sends are always synchronous.

The first two rules hold *even* when the message is found in a different parental object. Suppose a message is sent to a passive object c whose parent is an active object behaviour. Any message sent to c that needs to be delegated to its parent – because c does not implement the message directly – will eventually execute in a call frame under that parent. Since the parent is the *behaviour* of an active object, we know that it is serialized, so a lock will be acquired on the parent before executing the method. If the active object was busy computing, the sender of the message will have to wait until the active object is finished so that the child can use its behaviour. Only then can the delegated message acquire the lock and start computing. Applying rule 3, the method invocation will happen purely synchronous, even though the method found is one implemented by an active object behaviour.

5.6 Conditional Synchronization

As mentioned in chapter 3, conditional synchronization is an important part of any concurrent programming language. It allows two concurrent processes to synchronize based on some condition. In section 3.5.2 we have made a survey of existing

mechanisms to allow conditional synchronization. In a first attempt, we have tried to allow conditional synchronization *without* adding new features to the language. This was made possible by a special usage of promises. Subsequent sections will discuss some of our ideas on using different conditional synchronization mechanisms.

5.6.1 Synchronization via Promise Chasing

It is possible to add conditional synchronization to cPico without any new supporting constructs by exploiting promises. To support conditional synchronization, we notice that what we would really want is to "delay" an incoming message send. How can this be achieved? One way of delaying an incoming message is by *rescheduling* it. This is possible by performing an asynchronous self-send to `activethis()`. Consider a small example to illustrate this delaying scheme.

```
stack(siz)::{
   stk[siz]:void;
   top:0;
   push(x)::
    if(top=siz,
       activethis().push(x),
       stk[top:=top+1] := x);
   pop()::
    if(top=0,
       activethis().pop(),
       stk[(top:=top-1)+1]);
   activate() }
```

The example defines a bounded stack, in which a `push` request will have to be delayed whenever the stack is full and a `pop` request should be delayed when the stack is empty. The code actually implicitly contains two "guards", namely `top=siz` and `top=0`. Whenever such a condition holds, the processing of the message can be delayed by rescheduling the request with the same arguments. Thus, if the stack is full, it can use its built-in queue as additional storage space for `push` operations that cannot be processed yet.

Things become a bit more complicated when looking at the `pop` operation. Whereas `push` will usually be called without caring about a return value, `pop` is called for retrieving a value from the stack. That is, it will be called using code like `x: s.pop()`. Notice that `x` will be bound to a promise that will be "fulfilled" whenever the `pop` message finishes executing. This is problematic if `s` would be empty. In that case the fulfillment of this promise should be delayed because `pop` cannot be executed yet. In cPico, this "delay of fulfilling a promise" can be achieved by using a special property of promises. They are able to become fulfilled by *other* promises. That is: the value that will be placed in a promise placeholder

can *again* be a promise. This is also sometimes called promise or future *chasing* (Feeley, 1993). Some languages (including cPico) allow for this. Other languages require promise or future fulfillment to be *strict*: a promise can only be fulfilled by some determined value.

This chasing of promises is also inspired by ACT1's *Guardian* actors (Lieberman, 1987). This type of actors can be used for conditional synchronization in a manner similar to ours. That is, conditional synchronization is achieved by delaying the response sent to a customer actor. The "reply address" is remembered and a reply will be sent whenever the conditions are right. In our terminology we can take this "delay" of a promise very literally. That is, when an object asks another active object to do something, it receives the promise (to be taken literally!) that the work will be done. Now imagine that the active object is in no position to carry out the request right away. In that case, it has no other option but to fulfill its promise with *another promise*, namely "that it will carry out the work some time later on". The active object can use this promise chasing to delay his task.

Continuing the earlier stack example, observe that a call to `s.pop()` returns a promise p that, in the case of an empty stack, will be fulfilled by the value of `activethis().pop()`. Since this is an asynchronous message send (to the active object itself), it evaluates to a promise q. But this promise will also be the return value of `pop`, so p will effectively be fulfilled by q. If the user of `s` were to use the promise p, this promise forwards any operation to its determined value (which is q). It is then q that will make the caller block if it is not yet determined (meaning the `pop` message is still not processed). Figure 5.3 visualizes the situation for two recursive delays. Each piece of code evaluates to the promise depicted below it. p and q are considered fulfilled promises.

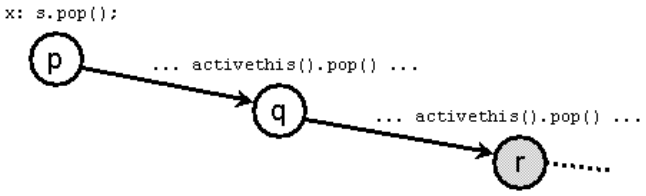

Figure 5.3: Promise Chasing for Conditional Synchronization

Although at first sight it might seem nice to have conditional synchronization without any special construct besides promises alone, we have to admit that this mechanism is more eccentric than useful. The mechanism is not always that easy to program since one must continually think about the underlying promises and make sure that the resend is the last expression in the method body. Moreover, it is not a good way to perform conditional synchronization as it suffers from *busy wait*: if the stack is empty, it will continually lead to rescheduled `pop` messages, forming an increasingly growing chain of promises (i.e. the synchronization condition will continually be polled). These reasons have lead us to consider more sophisticated

conditional synchronization schemes.

5.6.2 Call-with-current-promise Synchronization

A second method of achieving conditional synchronization in cPico is to build upon our promise model introduced into the previous section. This approach tries to alleviate the aforementioned problems in two ways. First, it tries to attach less stringent conditions to work with conditional synchronization than those imposed by the call/return context. Up until now, promises were only created for an asynchronous method call and fulfilled only by the method's return value. This forced us to tail-recursively *reschedule* messages if we wanted to delay promise fulfillment. Second, we want to get rid of synchronization through busy wait. A more efficient scheme is wanted.

We have dubbed this conditional synchronization method *call-with-current-promise*, analogous to Scheme's *call-with-current-continuation*. The reasons in drawing this analogy will be explained in the following section. In essence, we introduce two new natives to work with promises *explicitly*, whereas up until now promises have always been passed around "behind the scenes" by the interpreter. One native is used to grab the "current promise". By "current promise", we mean the promise that must be fulfilled by the currently executing code. Any running cPico code serves the fulfillment of some promise. Recall from section 5.3.4 that we also model top-level source code evaluation as asynchronous method invocation. Thus, there is always some promise to fulfill, even when not working with active objects explicitly. The second native we introduce allows for *fulfilling* a "captured promise".

The delay native The native `delay` allows the cPico programmer access to the hidden promise that will be fulfilled by the currently running method invocation. One can imagine that it is defined as:

```
delay(exp(promise)) :: exp(reify(<real promise>));
```

That is, `delay` is a native much like `call` that evaluates its only parameter, where this parameter may make use of a variable called `promise` which will be bound to a reified version of the current promise. This gives the programmer the ability to *save* the promise in some data structure and fulfill it at a *later* point in time. `delay` has another – equally important – semantics and usage. Its typical use is to *prevent* the current promise from being fulfilled by the value of the last expression in a function body. That is, we want to short-circuit the return value of the function. This is why `delay` does not evaluate to *any* value: it terminates the current computation immediately and leaves the current promise *unfulfilled*. When evaluating

```
{ x: void;
```

```
  display("before");
  delay( x := promise );
  display("after") }
```

only `before` will be printed. Control will return immediately after evaluating `delay`'s expression. This will leave the current promise bound unfulfilled to `x`.

The fulfill native As mentioned before, the only operation defined on a reified promise is *fulfilling* it. This is done through the `fulfill` native, taking a promise and a value as arguments. `fulfill(p,val)` will fulfill p with val if p is a promise. Armed with this operation, we can fulfill the captured promise of the previous example stored in `x` with some value any time later on.

To see how these natives allow for conditional synchronization without busy wait, consider the following example:

```
obj() :: {
  item : void; waiting : void;
  put(elt) :: { item:=elt;
                if(!is_void(waiting),
                   fulfill(waiting, item)) };
  get() :: if(is_void(item),
              delay(waiting:=promise),
              item);
  activate() }
```

This code transcript defines an active object where a `get` message is explicitly delayed until a `put` message arrives, without resorting to busy waiting. When a `get` message arrives, either `item` is returned if a `put` was already processed. Otherwise, the promise fulfillment is explicitly delayed and the promise is stored in `waiting`. The `put` operation verifies whether someone is waiting for the item and if so, fulfills the outstanding promise with its argument.

Reified promises We should still explain what exactly constitutes a "reified promise" and how exactly it differs from a real, hidden promise. A reified promise is actually nothing more but a wrapper around a real cPico promise. Whereas a real promise tries to "forward" all of its operations to its underlying value, a reified promise only allows `fulfill` operations to take place. If the promise is otherwise "touched" (by invoking any other operation on it), an error is raised. The reasons in doing so are mainly related to *safety*. Subtle bugs might creep into programs which would be hard to detect if the captured promises were as transparent as their real counterparts.

Consider the code `delay(1+promise)`. Although it is a contrived example, it illustrates the core problem: the `+` operation will touch a promise which the executor ought to fulfill itself. The addition is a strict operation and thus needs

its arguments to be resolved. This will make the evaluating active object wait for the promise to be fulfilled. But it is the active object itself that ought to fulfill this promise, to politely answer to the request of a client object. This is obvious in this small example, but recall that a promise can be stored without touching it in some data structure and then use it transparently in a context outside of a `delay`. Using reified promises, this code will result in an error instead of a hard-to-debug deadlock. Reified promises can only be usefully employed by a `fulfill` native.

Representing captured promises by special wrapper objects has its disadvantages, however. First of all, whereas real promises *become* their value once fulfilled, this is not the case with the promise wrappers. That is, if `x` is bound to a captured promise, and we evaluate `fulfill(x, 1)`, then `x` will still point to a reified promise (albeit a fulfilled one). Evaluating `x` will *not* result in 1. Second, since wrappers always clash with identity, the equivalence operator ($\sim$) will compare wrapper identities and not their underlying values. These "disadvantages" are not much of a problem since reified promises are *not meant* to be transparent, in contrast to their base-level counterparts.

Synchronous invocation semantics It is important to stress that an asynchronous method invocation as always having an associated promise to fulfill. This means we can rewrite every asynchronous method invocation of `m(args)::body` using `delay` and `fulfill` as:

```
m(args) :: delay(fulfill(promise, body));
```

We can then raise the question what the semantics are for `delay` and `fulfill` for a synchronous method call. We could *also* associate promises with synchronous method calls, with the additional semantics that the caller *always* immediately touches the promise, so that it will block until the method returns. Thus, each synchronous method call `o.m(args)` would be replaced by `touch(o.m(args))`, where `touch(exp)` waits for promise fulfillment and returns the fulfilled value instead of the promise. Although technically it would be possible to implement such concepts without resorting to using real promises, we currently do not assign any new, special semantics for synchronous method invocation. This type of invocation does *not* involve any promises. The semantics of `delay` is then to capture the promise of the asynchronous method invocation in which this synchronous method invocation is spawned. The `delay` will then also return immediately to the "return address" of the asynchronous invocation, meaning that it can non-locally jump over a large number of synchronous method invocations in one step. All locks taken during this asynchronous method invocation will be properly released.

5.6.2.1 Analogy with call-with-current-continuation

The reader acquainted with Scheme's powerful *call-with-current-continuation* construct will undoubtedly have drawn the parallel between `delay` and `call/cc`. In

this section, we will motivate our reasons for naming this synchronization method call-with-current-promise. For details on `call/cc`, we refer back to section 2.5.3.2.

We start the analogy by noting that Pic% (and Scheme) can manipulate *continuations* in a structured way through the natives `call` and `continue`. "call/cp" allows for structured manipulation of *promises* through the natives `delay` and `fulfill`. Both continuations and promises are implementation-level concepts, normally not directly visible at the base level. Both crucially determine the control-flow of a program. One can say that both concepts are juggled around "behind the scenes" and that the provided natives allow us "to grab hold" of these invisible entities.

The analogy does not end there. The reason why we have to provide access to the current continuation through a native like `call` is that sometimes we want to *avoid* capturing some computations. For example, when evaluating `1+call(x:=cont;2)`, we will not capture the assignment to `x`. This is necessary to allow for the manipulation of the continuation to happen "outside" of the current continuation. Similarly, our `delay` native temporarily *avoids* fulfilling the current promise, by immediately returning without return value. This is an important part of the semantics of `delay`.

The parallel between `continue` and `fulfill` is obvious: both allow for control-flow to continue and both are used to manipulate the captured concept. Also, a continuation is represented by a wrapped *environment*. Promises are represented by a wrapped "reified version" of the underlying transparent promise. To end the analogy, we should note that "call/cp", just like call/cc should be used to build more high-level abstractions. One often finds applications of call/cc in building coroutine or exception handling abstractions. Our claim is that call/cp should be used in a similar fashion, but for synchronization constructs. We will give an example ourselves in the next section.

5.6.2.2 An Example: Rendez-vous

The call-with-current-promise synchronization scheme obviously needs a more detailed example to illustrate its proper use. We will therefore discuss an implementation of *rendez-vous*. We will construct a `rendezvous` object through which two arbitrary active objects can perform a handshake, leading to implicit synchronization and value exchange. This functionality is surprisingly simple to implement using call/cp, whereas the solution using promise chasing was rather clumsy and inefficient. The code for implementing the `rendezvous` object is defined below:

```
rendezvous() :: {
   firstVal : void;
   first    : void;
   sync(val) ::
     delay( if(is_void(first),
```

```
                { first:=promise; firstVal := val },
                { fulfill(promise,firstVal);
                  fulfill(first,val)} ));
   activate()
};

rv?!val :: touch(rv.sync(val));
```

Two active objects can synchronize by invoking `rv?!val` on a third `rendezvous` object. The `?!` operator will block on the promise of the `sync` method. Only when both synchronees have invoked this operator will the value of it evaluate to the value passed by the other synchronee. Note that since we are dealing with method activation on active objects, concurrent method invocations are properly serialized. Race conditions are thus ruled out. The first synchronee finds the variable `p` initialized to void and therefore knows it is first. It will leave its own promise unfulfilled and stores it in `p`. It also leaves behind its value in `v`. The second synchronee will then retrieve this promise and perform a simple data exchange by fulfilling the promises with one another's values.

Note that `rendezvous` needs to be an active object because we have explicitly chosen not to incorporate promises for synchronous method invocations. This way, the `delay` actually "makes sense". Also, in a well-structured program, `delay` should always be the last expression in a method body. A complete version of the `rendezvous` object would also perform a reset of the instance variables after the swap operation. This way, the object is reusable for multiple synchronization attempts.

5.6.2.3 Expressing OR-parallel Evaluation

We have not yet mentioned what semantics to give to a promise that becomes fulfilled more than once. Three approaches are possible: we could throw an error upon all fulfillments except for the first, we could ignore subsequent fulfillments or we could enqueue subsequent fulfillments. The third approach is taken in the language ABCL (Yonezawa et al., 1986; Taura et al., 1994). Using this approach, a promise or future becomes essentially a queue. Although this approach allows for very expressive communication patterns between callers and callees, we have not opted for this approach since it no longer makes futures transparent. If a future can have more than one value, the caller has to explicitly *extract* values from the future, diminishing all benefits of transparency. We have opted for the second approach of *discarding* subsequent values, because it keeps promises simple and because it does not really introduce any problems.

The semantics of *allowing* multiple promise fulfillments actually gives rise to a very elegant solution to OR-parallel scheduling (Taura et al., 1994). An OR-parallel scheduler starts a number of tasks in parallel and waits for the results of only *one* of the tasks, usually the results of the *first* task to complete. This is called

Eureka Synchronization in (Taura et al., 1994). Lieberman (1987) introduces a similar construct called RACE, which is the future-based version of LISP's CONS. RACE will evaluate all of its elements in parallel and construct a list of the values in the order in which they have finished computing. A process accessing such a list must block whenever it is still unknown what the next element in the list will be.

We have implemented a simple version of Eureka synchronization which will evaluate a number of expressions in parallel and returns the value of the expression to finish evaluating first. In *speculative computing*, such constructs are used frequently to start various algorithms for the same problem and to return the value of the algorithm that finishes first. Some algorithms will be beter suited than others depending on circumstances which may be hard to capture. Of course, in a decent implementation, we would be able to *stop* any running computation when the "winner of the race" is known. A simple OR-parallel scheduler based on multiple promise fulfillment is shown below:

```
evaluator() :: {
   eval(promise, exp) :: fulfill(promise, touch(exp()));
   activate() };

scheduler() :: {
   race@exps() ::
       delay(
        for(i:1, i<=size(exps), i:=i+1,
            evaluator().eval(promise, exps[i:=i+1])));
   activate()
};

scheduler().race(
  { sleep(random()); display("nr1 arrived"); "nr1 won" },
  { sleep(random()); display("nr2 arrived"); "nr2 won" },
  { sleep(random()); display("nr3 arrived"); "nr3 won" })
```

When evaluated, this code excerpt creates a `scheduler` active object which will process a `race` message, taking an arbitrary number of expressions. The scheduler then starts up an evaluator active object to evaluate each expression and instructs the evaluator to fulfill the promise of the `race` method. The example creates three "runners" which arrive after sleeping an arbitrary amount of time. The runner that "arrives first" will fulfill the promise with a text string. This string will then be printed onto the screen.

5.6.2.4 Related Synchronization Schemes

It is interesting to determine under which conditional synchronization scheme (listed in section 3.5.2) we can categorize call-with-current-promise. Not surprisingly,

call/cp fits the continuation-based conditional synchronization schemes (section 3.5.2.5), albeit in a slightly different context. PScheme's (Yao and Goldberg, 1994) ports are really extensions of continuations to a parallel context, whereas our promises are merely "synchronization points" on which a continuation can block. In PScheme, `throw` can be used to send subsequent values through a port which will be enqueued and which can trigger the attached function multiple times. This is not so in our context: we can fulfill promises multiple times, but each fulfillment will not "trigger" the process that was blocked to re-evaluate some code. In this regard, `call/cp` is very similar to `call/sp`, because a *single* port will also discard all but the first value thrown into it.

We can also draw the parallel with ACT1's Guardians (Lieberman, 1987). In ACT1, each message has an associated implicit *continuation*, to which the result of method invocation is sent. This should not be surprising as promises and continuation actors are closely related. Guardian actors are special in that they can "grab" their continuation actor and explicitly send them the result. Lieberman (1987) also uses this method to *delay* the reply of the result to the sender. In ACT1, guardians are serialized, just like cPico active objects, so a delayed method invocation has to terminate to ensure that other messages can be processed.

Another analogous concurrency control mechanism is noted in ABCL/f (Taura et al., 1994). There, every method invocation also carries along an implicit future to be fulfilled by the last expression in the method body. The equivalent of call-with-current-promise in ABCL/f – which employs a syntax based on Common Lisp – is expressed as:

```
(defun f() :no-implicit-reply t :reply-to future
   ...
  (reply value to: future)).
```

The optional argument `no-implicit-reply` ensures that the last expression will not "fulfill" the future, while `reply-to` can be used to get a reference to the implicit future. The combination of both constructs is the equivalent of our `delay` native. The `reply` special form can be used to send a result back to a future, the equivalent of `fulfill`.

In conclusion, we note that `call/cp` is powerful enough to support conditional synchronization without resorting to busy wait. However, just like PScheme's multi-ports, `call/cp` is low-level and hardly reusable. It is subject to the inheritance anomaly since extensions of an object cannot deal with synchronization code in a modular way. The mechanism is only practically applicable when used to create more high-level synchronization constructs. The problem is that such high-level synchronization constructs are usually not cleanly integrated in the language, they are merely user-defined abstractions. A high-level synchronization mechanism that would be integrated within the language would better facilitate the construction of concurrent programs.

5.7 The Pic% Model Reconsidered, a Second Time Around

The cPico model that was introduced in 5.2 diverges on several important points from the Pic% model which served as our starting point. Most importantly cPico has ruled out dynamic scope in favour of lexical scope. Since such decisions should not be taken lightly or for superficial reasons, we will show how concurrency issues have lead us to the cPico model, starting from the Pic% model. The Pic% model was already introduced profoundly in section 2.5.3, but we recall some of the essential features here.

- Pic% unifies its environment model with its object model. There is no difference between a (conceptual) call frame (extension) that contains bindings for arguments of a function on one hand and an object on the other hand.
- Objects can be constructed using constructor functions (De Meuter et al., 1996), and object-based inheritance is achieved using nested mixin methods (Steyaert et al., 1993).
- Pic% unifies the concepts of functions and methods and makes these entirely first-class.
- Finally, Pic% reintroduces dynamic scope (D'Hondt and De Meuter, 2003) to encode an intelligent code sharing technique expressively, shunning the multiple inheritance solution used in Self (Ungar et al., 1991).

In section 5.3.1 the concept of a process and an object were merged into an active object, but this particular design choice has no effect on the issues that will be discussed in this section. Since the use of parent sharing in a concurrent and distributed setting is the core concept under investigation it would not make sense to rule out that two active objects can manipulate children of a shared parent. However, we *do* want to avoid concurrency problems such as race conditions, which were introduced in section 3.3.

Another very important issue in cPico is that transparency of some concurrency issues should be strived for. cPico should impose a minimal overhead when adapting a sequential program to a concurrent one. Since transparency is important it seems natural that we have opted for *implicit* concurrency control, reflected in transparent promises and implicit locking schemes. One design of such an implicit locking scheme is discussed next.

5.7.1 Automatic Locking

Integrating an implicit locking scheme in a highly flexible language such as Pic% is very hard. One of the main problems is deciding what exactly needs to be locked. Recall that Pic% objects consist of a list of variable bindings and a list of constant bindings. These objects can be connected in a parent-child relation. The visibility

of an object's methods extends up to the global root dictionary in Pic%. Thus, the scope of a parent is subsumed by the scope of its child.

How can the granularity of a lock be defined using such scope rules? Objects whose data is used by a method call on itself *or one of its children* should always be locked to avoid the problems explained in section 3.3. At first sight one may be tempted to explore an incremental locking scheme, which just locks all parents when a method is invoked on a child. Such a scheme is illustrated by figure 5.4. When a function f is called, the object in the hierarchy that executes f is automatically locked. Lookup for the variable x need not be delegated to an unlocked object, so no additional locks have to be taken. To find the variable y, lookup must be delegated to some unlocked frames, which are locked prior to starting lookup in the object. All locks are simultaneously released at the moment the function returns.

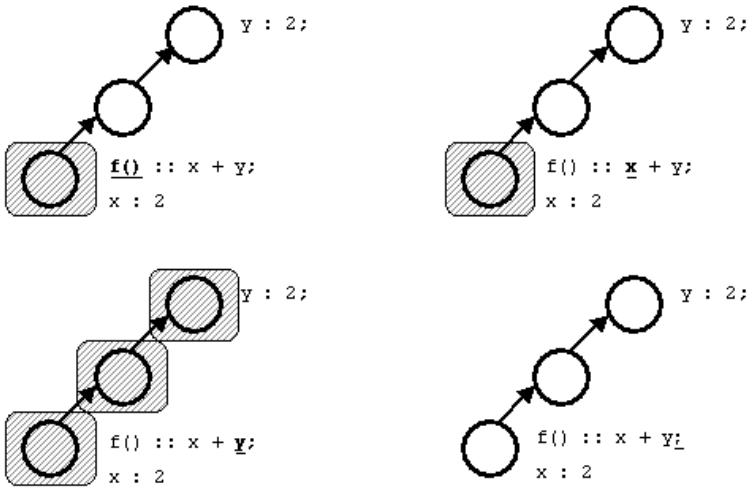

Figure 5.4: Incremental Parent Locking

Whereas such an approach is feasible, it does require taking a whole range of locks which is very expensive to do for each variable lookup that needs delegation. Moreover, this scheme imposes too much restrictions on concurrency, since "intermediate" objects will sometimes be locked even though this is not needed. Yet another disadvantage is that these locks are not taken atomically and therefore introduce a plethora of possibilities for deadlocks to occur.

Ideally only *one* object should be locked upon a method invocation, rather than a whole hierarchy. This is not possible when employing Pic%'s scoping rules since child methods have unlimited access to all variables that are defined along the parent hierarchy. To restrict the amount of data that needs to be locked, method scope must be restricted to deny access to variables that are not local to an object. Such a restriction was added to cPico. The changes compared to Pic% are discussed in the next section.

5.7.2 Call Frames Versus Objects

The problem illustrated in the previous section is that any method can access variables that may be defined anywhere in the parent hierarchy. Since these variables could be *shared* by multiple children and thus *shared* by multiple passive objects, race conditions must be prevented. This requires a lock on the entire delegation chain. A solution to reduce the granularity of locking is to change the method's scope from **protected** to **private** (in e.g. Java or C++ terminology). This way, only the variables of the local object can be accessed, ensuring that a lock is only acquired on that particular object, and not on any of its parents.

To change the scope of a method execution, a difference must be made between real "objects" and mere "extensions" (call frames) in which method bodies are evaluated. To see why, notice that a call-frame is always an extension of the object on which the method is invoked. This enables proper scoping rules, as the method can now "see" the instance variables of the receiver. The most important distinction that has to be made between objects and call frames is this: whereas call frames implement delegation for **both** constants and variables, objects will **only** delegate lookup for constants, not for variables. This ensures that a method can still use variables of its extended object, but no longer those variables of any of that object's parents.

A distinction between objects and call-frames is quite natural. Whereas both consist of a constant and a variable part containing a number of bindings, and both have a pointer to the "next" object or frame, objects can have more properties than call frames. In a concurrent setting, for example, objects can be locked, while call frames cannot. A small example will illustrate the new scoping rules implied by the new delegation semantics.

```
parent(x) :: {
   getx()::x;
   child(y) :: {
      m(z) :: getx() + y + z;
      capture()
   };
   capture() }
```

Figure 5.5 shows a code snippet and the corresponding layout of the objects in memory. The frame is visualized by a dashed border. Recall that arguments to constructor functions in Pic% are also variables of the constructed object. The variables are no longer "protected" in the Java-sense: in the original Pic% model, `child` would have been able to access x without any syntactic annotations. In cPico, even a child cannot access the variables of its parent: when writing `x`, x will be found only in the current call frame *or* in the receiver when it is a variable. The call frame will delegate variable lookup if it cannot find x, but an object will not. Constants however, are still delegated by both call frames and objects. Constants do not pose problems in the context of concurrency since they are read-only. Although

`child` has not defined `getx` in the above example, it can still access it because `child` will delegate to `parent`. Notice that m can use y and z freely (they are local to m or to `child`), while it has to access x through a provided getter function, just like any other external object.

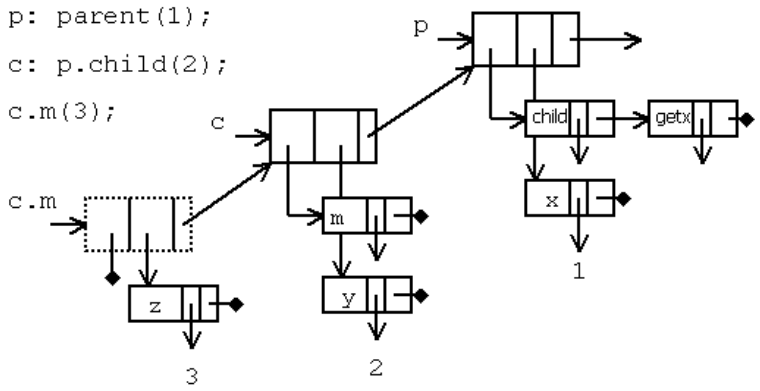

Figure 5.5: Pic% Object and Frame representation upon method invocation

When executing m, `getx` will first be executed. Due to Pic%'s dynamic scoping, the call frame for this method will be constructed below the call frame of `m`, not below the parent object. This introduces the problem that during the execution of `getx`, x will not be found. This is due to the fact that lookup for x must traverse a child object which will not delegate lookup of variables. Nevertheless when a method is defined in a parent, it should be able to access the lexically visible variables of its defining object. Dynamic scope makes this impossible. If static scope is employed, the call frame of `getx` is an extension of `parent` and the intended semantics of the program are honoured. Other implications of the use of dynamic scope as defined in Pic% will be discussed in the next section.

5.7.3 Dynamic Scope

Dynamic scope is one of the most remarkable features of Pic%. It allows for the sharing of code in a more natural way than e.g. the traits scheme of Self Ungar et al. (1991). Nevertheless dynamic scope also raises some problems from a software engineering point of view, especially when combined with objects with protected variables, as was already pointed out in the previous section. The *dynamic scope* as it is currently conceived in Pic% seems to reintroduce at least part of the problems observed in the early versions of LISP.

In cPico we have chosen to avoid aforementioned problems by reverting to static scoping rules. Pic%'s natural code sharing is maintained using an adapted lookup scheme, explained in detail in section 5.8.2. The difference between static and dynamic scope is that statically scoped functions or methods' call frames will be extensions of the environment of definition, while dynamically scoped methods'

call frames are extensions of the environment of invocation. In cPico, there are three ways to access a slot `x` inside a method body:

- One can use `x` to denote a variable or function statically. In the original Pic% model, this variable was also looked up dynamically. In cPico, x will be looked up in the method's or function's local variable records or in the static (lexical) object of the method.
- An explicit self-send like `this().x` will result in dynamic lookup. This is logical considering late binding of self. Super sends or delegation will leave the `this()` receiver variable unchanged. This method still allows for variable overriding. One drawback of this is that in Pic%, only constants can be accessed through qualification via the dot-operator. This means that overridden variables cannot be accessed through this mechanism. We have not complicated the semantics of message sends by checking whether the receiver is `this`, and attributing special behaviour to this case, especially since scope functions alleviate this problem.
- A third way of referencing a variable or a method is through super-sends using `.x`. This will initiate method lookup in the *lexical* parent, not the parent of `this()`. Due to static scope, the static parent is in fact the parent of the object to which the call-frame is attached.

Introducing static scope in combination with the lookup schemes discussed above allows the programmer to restrict lookup to the lexical environment, by using regular function calls of the form `f(args)`. This introduces two additional benefits that can be important in a distributed setting, namely *security*, and *safety*. We will get back at these benefits in section 6.10.

We are not the only language designers having scaled down a dynamically scoped system to a statically scoped system for reasons of introducing concurrency. Multilisp, being an offspring of the originally dynamically scoped language LISP has also opted for static scope. The reason in doing so is explained in (Halstead, Jr., 1985, p. 509):

> *Lexical scoping [...] decouples the choice of variable names in a procedure P from the choice of free variable names in other procedures that call or are called by P and thus promotes modularity more effectively than the traditional Lisp discipline of dynamic scoping [...]. Furthermore, the usual optimized implementation of dynamic scoping by "shallow binding" does not adapt gracefully to a multitask environment where various tasks running in the same address space may have different values for the same variable. Implementation of lexical binding by means of a static chain of environments continues to work well.*

5.7.4 Object Creation

Due to the separation between objects and call frames in cPico, object creation has to be reconsidered. The *only* object constructor in the language is `capture()`, which used to "inject" the current call frame into the current object hierarchy. All other natives that can be used to construct new objects, such as for example `clone` and `activate` will implicitly perform a capture first.

5.7.4.1 Capture semantics

In the original Pic% model no such explicit transformation is needed since the call frame *is* the captured object. By distinguishing between objects and call frames, `capture()` has to return an object, and so it has to explicitly transform a call frame into an object. Converting the most closely nested frame is not enough, however. `capture()` will have to transform all frames starting from the current call frame up to the first object it encounters along the linked list of frames, determined by the scoping rules. An example will illustrate why this is the case:

```
m() :: {
   x :: 1;
   n(y) :: { method() :: x+y; capture() };
   n(2)
}
```

This example defines a function m that contains a nested function n. When `m()` is called, a new call frame will be created. In this frame a function n is defined and subsequently called. Using lexical scoping rules for method application, the activation record for n extends the activation record for m and y will be bound to 2. This way, both x and y will be visible inside of n, as expected. n appears to be a constructor function that returns an object. If we adhere to the new definition of `capture`, which will capture and transform all call frames up to the first real object, then the new object will contain `method`, y *and* has a parent containing x and n. If only the most closely nested call frame was captured, any reference to x or n would be lost, thus upon calling `method` x would not be found. `capture` recursively traverses the call stack and transforms frames into objects one by one. The result of the capture is the transformation of the most closely nested call frame.

5.7.4.2 Dynamic Versus Static Mixins

Another way to create new objects is by calling mixin methods on existing objects. The difference in semantics of function calls versus message sends invoked through `this()` allow for the implementation of two types of mixins. Since any invocation through `this()` results in dynamic scope, there is a big difference between invoking`m()` and invoking `this().m()`. Let us illustrate this with a well-known example to demonstrate linearized inheritance through mixin classes in CLOS (Budd, 2002, chap. 13):

```
staticPerson(name) :: {
   title()  :: name;
   asDoc()  :: {
      title() :: "Dr. "  + .title();
      capture() };
   asProf() :: {
      title() :: "Prof. "+ .title();
      capture() };
   capture() };
dynamicPerson(name) :: {
   title()  :: name;
   asDoc()  :: {
      title() :: "Dr. "  + .title();
      this().capture() };
   asProf() :: {
      title() :: "Prof. "+ .title();
      this().capture() };
   capture() };

staticPerson("Einstein")
   .asDoc().asProf().title() = "Prof. Einstein";
dynamicPerson("Einstein")
   .asDoc().asProf().title() = "Prof. Dr. Einstein";
```

The main difference between the examples is the usage of `capture`. In `staticPerson`, `capture()` will create a statically scoped view, while in `dynamicPerson`, `this().capture()` will create a dynamically scoped view. As the example shows, both are expressible in our model. Dynamically scoped mixins are rather powerful because they effectively allow the same kind of "multiple inheritance" as in CLOS, namely linearized inheritance. This allows for mixing in different kinds of behaviour from all layers of the hierarchy to create rich compound objects. In (Lucas and Steyaert, 1994) it is shown that such hierarchies are more structured than those that can be achieved with conventional inheritance. The difference in structure between the static and the dynamic person objects is outlined visually in figure 5.6.

5.7.5 Natives

cPico reconsiders evaluation rules for some of the existing Pic% natives. Pic% natives are only considered as functions that should plainly be applied instead of sent to an object. There is one notable exception, being `o.clone(upTo)`. However, more natives exist whose semantics should be adapted when invoked through a message send. These natives are called *reifier natives* in the implementation. One example of such native is `eval`. Invoking `o.eval(exp)` will evaluate `exp`

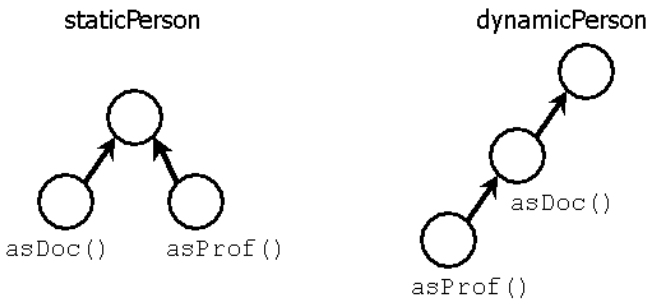

Figure 5.6: Static versus dynamic functional mixins

in the context of the receiver. Other such natives are the aforementioned `clone`, `capture`, `this`, ...

The semantics of sending such natives to objects will be clarified by explaining the previous example on dynamic mixins. Consider the dynamic mixin creation using `this().capture()`. First, the native `this` is looked up and applied, yielding the current receiver which will become the parent of the new view. Lookup for `capture`, will once again yield a native[3]. A message send of the native `capture` to `this()` is subsequently evaluated. Since `capture` is a reifier native, it is treated specially and rather than just triggering the native's "apply" behaviour, its "send" behaviour is executed. A `capture` native sent to an object will create an object from the "current call frame" as explained above. The parent of the newly constructed object will point to the *receiver* of `capture` instead of the lexical environment of the mixin implementor.

Special attention is payed to one more native, namely `super`. When introducing the different ways of invoking a function, super delegation was also mentioned using `.m()`. Such invocation is *not* the same as `super().m()` since in cPico `super()` denotes a first class object. Therefore, `.m()` features late binding of self, while `super().m()` will not.

Having first-class `super` objects is an uncommon language feature, since most languages only provide a *super* keyword referring to a "super object" to which method lookup can be delegated. In class-based and concatenation-based languages, the super cannot be seen as a real object because delegation implies late binding of self. This implies that *"the self of super is not itself"* (De Meuter, 2004). *super* in such languages is most obviously not a true self-contained object and can thus not be seen as an entity separable from `this()`. In Pic%, being a delegation-based language, a parent or "super" object is an object in its own right, having a separate existence. A message to the parent (using `super().m()`) is a plain message send and *not* delegation, featuring *no* late binding of self.

[3]It is assumed neither `this` nor `capture` are overridden in the object.

5.7.6 Summary

This section has given some more founded "delta with the Pic% model" as we have introduced it in chapter 2. In order to achieve intuitive program behaviour, which is especially important for complex programs such as non-trivial concurrent ones, the power and flexibility offered in Pic% were reduced in a number of ways. First of all, a clear distinction was made between a call-frame which has extended visibility to whatever lies above (be it another call-frame or an object). Objects, on the other hand cannot delegate lookup for variables. As such, child objects can only access the constant part of their parent, just like any other object. However, when writing programs it sometimes becomes inevitable that only a certain set of objects can access a parent's variables. This is actually the reason for making them delegate to that particular parent. Thus a child should actually have more rights than other objects. This topic has been dealt with in section 5.4.1 where we have introduced *scope functions* that alleviate this problem.

Furthermore our distinction between objects and call-frames has required us to reconsider both the use of dynamic scope, which is impossible to uphold if variables are made private. Another repercussion is that an explicit object creation native is needed that transforms a call-frame into a true object. Finally, a distinction between ordinary natives and reifier natives was introduced. Reifier natives exhibit some special behaviour if they are sent to an object. For a more formal overview of our changes to the language regarding static scope, we refer to appendix A.2.

5.8 Implementation

This section will go into some more detail on the implementation promises, lexical scoping and garbage collection of active objects. It is not our intention to go into the details of the specific Java implementation. For promises however, the Java implementation is unavoidable as it discusses how synchronization using promises is reflected in the underlying Java synchronization mechanisms. For static scoping, the emphasis is more on the abstract level, and can be interpreted independently from the implementation language. Slightly more formal semantics for this implementation can be found in appendix A.2.

5.8.1 Promise Representation

The goal of this section is to show how we have integrated the `Promise` data type in the cPico interpreter. In most languages making use of futures or promises, the representation is straightforward. In (Baker Jr. and Hewitt, 1977), a future is represented as a 3-tuple (process, cell, queue). In (Halstead, Jr., 1985), it is represented as a quadruple (lisp value, task queue, determined flag, lock). Lieberman (1987) also describes futures in terms of a value slot and a flag indicating determinacy. Our implementation closely resembles the one of multilisp. Conceptually, a cPico promise consists of:

- A cPico value, initially void. When the promise is fulfilled, this value will be assigned the fulfilled value.
- A boolean determining whether this promise has already been fulfilled.
- A lock to assure promise checking and promise fulfillment are atomic and mutually exclusive.
- A waiting queue in which processes that are waiting for the value of the promise are enqueued. Any process in this queue is *suspended*. When the promise gets fulfilled, it *resumes* all of its waiting processes. This is used to avoid busy waiting.

Notice that these are the "conceptual" constituents of a promise. In the Java implementation, a `Promise` class consists of only the first two attributes explicitly. The second two are really at the meta-level (that is, we use implicit Java locks and the Java waiting queue associated with an object's monitor). The `Promise` class implements *all* operations that any cPico value class must understand. All such implementations are of the form:

```
Returntype methodName(arguments) {
   return getValue().methodName(arguments);
}
```

This ensures that all operations are just forwarded to the underlying value and effectively makes promises invisible, *also* at the implementation level! The only operation which is not overridden in this way is the method `isPromise` which returns true, while it returns false for any other cPico value. This way, the implementation can still discriminate between promises and other cPico values. The method `getValue()` is of course the most important method of a `Promise`. The method first checks whether the current promise is fulfilled. If it is, it just returns the value in its value slot. The promise's role has then degenerated to a simple indirect pointer to the fulfilled value. If the promise is not yet fulfilled, the currently executing thread is enqueued in the waiting queue. This is done implicitly using `wait()`. The `fulfill` operation on a promise will assign the correct fulfilled value, will set the determined flag and will then "resume" all blocked threads in the waiting queue. This is done implicitly using `notifyAll()`. Because both `getValue` and `fulfill` are declared `synchronized`, they will execute atomically and mutually exclusive, guaranteeing consistent behaviour.

Notice that this method of implementing promises has the advantage that synchronization not only becomes invisible at the cPico language level but *also* at the Java implementation level. All synchronization is completely encapsulated inside one class. There is a catch in doing this, however. One can think of a promise as a "proxy" for the real value. As such, we have to be careful at the implementation level not to use downcasts or `==` equality operators. Recall the discussion

in section 4.2.5.1. Downcasts are as such avoided in the implementation by replacing them by method calls of the form `asDesiredValue()`, having return-type `DesiredValue`. Such methods will either return `this` if `this` is of type `DesiredValue` or raise an error if it is not. This offers the same functionality of downcasting but is much more safe, controllable and reusable.

Let us give an example of how implicit synchronization is achieved in the implementation. Consider the evaluation of `x+1`, where `x` yields a promise. Both arguments to the `+` operation have an abstract type which can denote *any* first-class cPico value. Because the `+` operation needs numbers, it needs to downcast these values to cPico's number type. By replacing this downcast by sending an `asNumber` message to both arguments, the thread executing `+` will automatically be stalled until the promise denoted by `x` is fulfilled. This happens as invisible at the cPico level as it happens at the Java level. This is an important property of an implementation, since it allows for writing many operations on objects, without ever having to worry about synchronization. If threads would have to be explicitly synchronized whenever a "strict operation" is performed, the implementation would be more susceptible to bugs, hard to maintain and less reusable.

5.8.2 Supporting Static Scope in Pic%

In section 5.2 the reintroduction of static scope has already been mentioned. These rules allow for more scoping control. However, we do not want to lose the conceptual ease of sharing code without resorting to concepts such as traits (see section 2.5.1.3). Such sharing is possible because functions do not contain their lexical environment. Because they are free from such a dependency, these functions can be reused by multiple objects. Yet, a lexical environment is essential to enable static scoping.

Our implementation reuses this sharing mechanism by employing a special lookup mechanism. To realize static scoping the lexical scope of a method must somehow be retrieved without storing it in the environment. To avoid reintroducing the problems that were identified in (D'Hondt and De Meuter, 2003), a scope for the function is provided *at lookup-time* instead of storing it at *definition time*.

Consider a message application of the form `o.m(a,b)`. The first step in the evaluation of this application consists of the lookup of m. The difference between Pic% and cPico is that in Pic% a *function* is returned, while in cPico a *closure* is returned by the method lookup mechanism. Argument evaluation and parameter binding will proceed in a similar fashion in both languages. In Pic%, the function will subsequently be applied and its body is executed in the context of (an extension of) the *current* environment. This results in dynamic scope, as the function will be able to access arguments visible at call-time. Because cPico will apply a *closure*, carrying a proper lexical environment, the function will get executed in the context of the paired environment. This enables static scoping rules.

It remains to explain how and when precisely the cPico closure is constructed. Since it is not *stored* together with the function, it must be *created* when the func-

tion is retrieved. Note that it is exactly this closure that would be the return value of the expression `o.m`. This shows that the the method lookup scheme readily supports the use of first-class methods. In fact, when applying a method, lookup will return a "first-class method" and will then immediately apply it.

Consider o to be the receiver from the above example. Lookup for `m` will start in the constant part of o. Consider a function m is *found* in the object p (possibly after delegation, in which case $o \neq p$). The closure, constructed at the moment the function m is retrieved, consists of:

- The function that needs to be executed. This function contains a name, an argument list and a body.
- The "current" dictionary, which is the *dictionary where* m *was found* (p in this case). Upon calling the associated function, the function body will be executed in an extension of *exactly this* dictionary. This is what re-enables static scope: "the environment of execution will be placed under the environment of definition". We consider the dictionary where a method was found to be exactly this "environment of definition".
- The late bound `this`, which corresponds to the *initial receiver* of the message. In the example, this is o, which is the place where self-sends[4] should start their lookup.
- The `super` dictionary, which is also kept static. In casu this means that the super dictionary is the parent of p. This ensures that super sends are evaluated in the enclosing lexical environment of the sender.
- The evaluator, which is active object to which the closure is tied. If o is an active object then the evaluator will be o, otherwise the active object that is currently executing the lookup is used.

In Pic%, only the receiver is stored in a closure. The added dictionaries support lexical scope, whereas the evaluator is used to keep track of which active object should execute the method. In short, the approach of attaching the static scope of a function at lookup-time allows for the function to be *shared* by multiple objects, since the function will always be looked up in the right receiver. Thus, the dictionary enclosed in the closure will always be the correct object. Functions are "dependency-free": they are immutable and can be freely shared, moved and re-entered between clones. To conclude, we refer to appendix A.2 for a slightly more formal definition of this lookup mechanism.

5.8.3 Garbage Collection of Active Objects

In his paper on the actor-based language ACT1, Lieberman (1987) pleads in favour of dynamic allocation of processes, just as would happen with e.g. LISP or Scheme

[4] Self-sends are invocations of the form this().m()

cons cells. Active objects, like cons cells, can be simply created "whenever needed", and should be reclaimed properly by a garbage collector to uphold the illusion that the user has infinite memory. Thus, the dynamic allocation and reclamation of processes should resemble that for simple data storage. The reasons for doing this are also similar: the explicit deletion or killing of processes is discouraged for the same reason explicit storage reclamation is discouraged. If two processes would communicate with a third process, and one of these processes kills the shared process, the other process would also be influenced by this operation. Lieberman calls it "explicit deletion of processes considered harmful" (Lieberman, 1987). This position is also defended by Baker Jr. and Hewitt (1977), who give the example of a database accessed by concurrent processes. If the processes must each lock the database before using it, and a process using the database is suddenly killed by another process, the database will remain forever locked, since the process has had no chance to properly release its lock. Making a process release all its locks upon sudden termination is no option: it would leave an object (or a database) in an inconsistent state.

Garbage Collection of active objects is also an important issue in so-called "eager beaver" evaluators (Baker Jr. and Hewitt, 1977) which evaluate their arguments using "future order" evaluation, as opposed to call-by-name or call-by-value parameter passing. With this type of parameter passing, each argument is evaluated in a separate thread concurrently. The result is a future which will make the called function block whenever it needs access to an actual argument that has not yet been evaluated. In such schemes, it becomes important to detect when processes are no longer useful. That is: a process should be terminated whenever it appears that the result it is computing is no longer required. Note that this is only true in a functional language. In an imperative language, it does not mean that a process may be killed just because its return value is not used, because of side effects. In (Baker Jr. and Hewitt, 1977), a garbage collection algorithm for such a functional evaluator is outlined.

5.8.3.1 Active Object Garbage Collection in cPico

Our implementation naturally maps active objects in cPico to threads in Java. One problem in doing this is that the thread of an active object will never die. That is, the thread driver will *endlessly* query its message queue for messages to process. Thus, Java will *never* be able to claim the thread since it never stops. We therefore need to perform a bit of garbage collection manually. The trick is to exploit the fact that each Java active object thread is coupled with exactly *one* Java object representing the first-class active object. Whenever this normal Java object is garbage collected, we know that no-one will be using the active object anymore. Therefore, we can safely stop its thread. This is possible in Java through the concept of a *finalizer*. This is a method that will be invoked by the garbage collector when the object is known to be garbage. Since each active object has a pointer to its associated thread, it is easy to shut it down from within this finalizer method.

By relying on the Java garbage collector to reclaim the active object, we must make sure that we do not create memory leaks by having pointers from the thread to the active object. If the thread would keep a reference to its associated object, the latter would never be reclaimed. Therefore, a link between the thread and its active object is *only* established when the thread is executing a method invocation on the object. We will now give an informal proof of correctness for our simple garbage collection scheme. We will show that our thread is only killed if and only if its queue is empty and it is guaranteed that no more messages will arrive.

First of all, note that because a pointer is kept from the thread to the active object during method execution, an active object can *never* be reclaimed while the thread is active. A thread can thus only be reclaimed whenever it is blocked on its queue. Whenever the thread is waiting for new messages to arrive (its queue is empty), it will have no more references to its associated active object. This means the active object can potentially be collected and memory leaks are avoided. Now consider that the active object is garbage collected and Java invokes the finalizer. At that moment, we have the strict guarantee that the thread is not computing, but blocked on his queue. Furthermore, it can be deduced that no more messages will arrive in this queue, since *no* other Java object has a reference to the active object anymore. The thread can thus safely be destroyed: it is blocked on an empty queue in which no other message will ever arrive.

Whenever a message is enqueued in a thread's queue, the message itself will contain a reference to the thread's associated active object. This ensures that an active object can never be reclaimed when its thread still has some messages left to process. Subtle race conditions that would kill the process could occur if such references would not be kept. If an active object is created, sent some messages and then all references to it are removed, the active object will first properly process all messages sent to it. In fact, we can safely state that an active object can only be reclaimed after all messages ever sent to it are processed.

5.9 Epilogue: Delegation Versus Synchronization

In their widely known paper on inheritance and synchronization Briot and Yonezawa (1987) discuss how concerns for delegation and synchronization can conflict with one another. We have identified this problem as well through our experiences in investigating synchronization in a *dynamically* scoped Pic%. One of the key features of what makes delegation so dynamic is that variables are conceptually also accessed through message passing, implying they can be easily overridden in child objects and that a parent can actually use "abstract variables" (analogous to abstract methods). Using dynamic scope, Pic% was equipped with such powerful features.

The conflict between distributing knowledge among objects (and manipulating such knowledge through delegation) and synchronization of concurrent access by multiple objects is illustrated in (Briot and Yonezawa, 1987) by the example of a `counter` object, having an `increment` method which updates its count by one.

The counter has to send `get` and `set` messages to its "variable" to manipulate it. Variable access happens through message passing to decouple variable values from their lexical context. The classical example of a ghost write can occur because an object can read the value of the counter variable before another object has had the chance of both reading *and* incrementing the variable. This is due to the fact that the read *and* write on the variable were not atomic. Accessing variables through message passing for atomic reads and writes only is not enough to safeguard entire methods from being atomic.

Similar problems occur when naively delegating to a parent to access or modify a variable. Since the parent object may be shared by multiple children, one must ensure that updates on its variables happen atomically. Again, when evaluating code like `n:=n+1`, it does not suffice to acquire a brief lock on the parent, read n, release it, add one to n and then re-acquire a brief lock on the parent to store `n+1`. Other children may access or modify the parent in between these actions. That is why we have introduced expressions like `super(x:=x+1)` which guarantee that the expression is executed atomically. Briot and Yonezawa (1987) also note problems with deadlock through recursive self-sends. Our model has prevented local (synchronous) self-sends from deadlocking using reentrant locks. Deadlock can still occur when dealing with multiple active objects working on the same object hierarchy.

Note that in Pic%, supporting delegation for constants has never introduced any problems, since they are immutable. In the context of concurrency, it was the implicit delegation of variables necessary for dynamic scope that caused aforementioned problems. Re-adopting static scope has been our solution to the problem. Thus, we have sacrificed some flexibility to incorporate more synchronization. This general tradeoff appears to be fundamental, as witnessed in (Briot and Yonezawa, 1987, p. 39):

> *[...] the attempts to increase the flexibility of inheritance by widely distributing functions and knowledge among objects complicates the synchronization issues. The tradeoff between the distribution for the sake of flexibility and the atomicity for the sake of synchronization appears to be fundamental in concurrent (distributed) computation.*

It appears that a simple delegation mechanism that incorporates atomicity, flexibility (such as variable overriding) and expressiveness is hard to construct. We have been able to solve some synchronization problems, but only at the cost of decreased flexibility. Our model strives for expressivity and achieves this in part due to its high transparency (automatic locking, no syntactical distinction between asynchronous or synchronous messages, lazy synchronization via promises). Atomic method invocation using serialized objects also avoids much of the problems with concurrent access. However, limited experience in programming in our concurrent extension has taught us that locks are too deadlock-prone. When a program deadlocks, transparency all of a sudden becomes a *burden* on the programmer

since he has to discover all *implicit* locks that have possibly lead to the deadlock. Debugging such programs is *hard* because locking is managed at the meta-level and is never really visible in the code.

5.10 Conclusion

In this chapter we have presented the language cPico, featuring a concurrency model for Pic%. The language is a middle ground between the actor and the thread paradigm. The former model is reflected in cPico's active objects and asynchronous message passing, whereas the latter is visible in such constructs as serialized objects and atomic method invocation. In order to support such a model we had to slightly reduce the flexibility of Pic%. The most drastic changes in this context were abandoning dynamic scope and undoing the total unification of the object and the environment model.

We have introduced the notion of active objects which are conceptually close to actors in the actor model. Our model does not incorporate behaviour replacement, and allows active objects to have true mutable state. Method invocation to active objects is also asynchronous, but it supports direct return values through the use of transparent promises. Because mixing active and passive objects in a parent-child hierarchy was complex and problematic, we have distinguished between the behaviour and the "active part" of an active object. Only the (passive) behaviour is used in parent-child hierarchies.

Because simple passive parent objects are allowed to be shared by several active objects, proper synchronization mechanisms were needed. A serialization of these objects, using reentrant locks was therefore introduced, guaranteeing atomic method invocation. We have extensively treated one particular sharing relation, called parent sharing and have advocated that if two objects share a parent through delegation, this parent provides extra security, safety and synchronization. Conditional synchronization was achieved through explicit control over promises using `call-with-current-promise`.

The most difficult part in developing a clean concurrency model proved to be the proper delegation of shared *variables* (as opposed to constants). Because data in a pure delegation-based scheme can be spread across multiple objects, synchronizing all these objects to ensure serializable methods is more problematic than in typical class-based schemes (Briot and Yonezawa, 1987). Also, implicit locking improves expressivity but can make programs hard to understand, especially when deadlocks occur.

Throughout the design of the concurrency model, *expressivity* and *simplicity* were strived for. Expressivity was illustrated in part by our "expressivity benchmark" in section 5.1.1, although it must be admitted that no exact scientific criterion exists that can adequately measure expressivity. A more elaborate example, illustrating a solution to the so-called *same fringe* problem can also be found in appendix B.1. Simplicity was achieved by allowing only passive-passive delegation

schemes and by reusing promises for conditional synchronization.

Concluding this chapter, our concurrency model must allow for a clean transition into a distributed context. Exactly how our concurrency concepts provide for a good distributed foundation will be explained in the next chapter. There, we will extend cPico with a distribution model, providing the necessary constructs to facilitate writing distributed programs.

Chapter 6

dPico: a Distributed Pic%

The previous chapter has introduced cPico, a concurrent version of Pic%. This chapter will build upon that language to explain dPico (distributed Pico), which extends cPico with a distribution model. In cPico, we have introduced concurrency through active objects. As we have argued before, the construction of a thorough concurrency model should make a distribution model with multiple collaborating virtual machines easier to develop. As will be shown, the distribution model introduces some intricacies of its own, requiring a redesign of a number of previously introduced language features.

In chapter 4 we have identified several language features required for a full-fledged distributed language. In section 6.1, we will clearly state which of these features are dealt with in this chapter and which fall beyond the scope of this dissertation. We will also illustrate how programs in dPico should be organized and it will be discussed how the new language both reuses and redefines concepts from cPico.

Continuing with section 6.2, two simple semantic rules will be introduced concerning object parameter passing across the network. These semantics will largely determine the unit of distribution of the model. However, clear as the new semantics may be, it will be shown that they provide inadequate support for *parent sharing*, a dominant characteristic of the cPico object model. A solution to this problem is presented in section 6.3 with the introduction of a new inheritance model featuring a parallel delegation structure to separate active from passive objects.

The dramatic change in the organization of object delegation requires us to review some of the language features of cPico. In particular, more controlled extension mechanisms are needed to achieve better object structuring. This is explained in sections 6.1.3 and 6.4. The introduction of a second – parallel – delegation hierarchy will also require a revision of method lookup and method invocation on active objects, which is done in section 6.5. This section also proposes a means to reduce network traffic in a distributed setting.

Subsequent sections will be more concerned with the distribution aspects of dPico. Section 6.6 will present the simple mechanism to acquire initial references

to remote objects. At that point, sufficient dPico knowledge is communicated to be able to canalize dPico's concepts in a small example application in section 6.7. Continuing with a renewed discussion on promises in section 6.8, we will focus on necessary changes to their representation and behaviour in a distributed setting. Section 6.9 will provide some pointers on how "continuation mobility" is achieved in the language. Indeed, some limited support for code mobility is present in dPico, due to the fact that any first-class value can be "moved" from one virtual machine to the other, and continuations (Pico environments) are first-class values. Although high-level well-integrated mobility abstractions fall outside the scope of this dissertation, we do not underestimate their importance to support "ambient-ready" applications. See (De Meuter, 2004) for a detailed treatment of such language constructs.

The dPico model has some characteristics that inherently promote security and safety. These features will be the topic of section 6.10. Section 6.11 will then put the dPico model into perspective by highlighting some of its limitations. Section 6.13 concludes the distribution model for Pic%.

6.1 Objectives

Chapter 1 has identified the concerns that need to be addressed by an "ambient-ready" programming language. Here, we will briefly review those required language characteristics and clearly state the ones that will be incorporated in dPico and those that will not.

Distributed object inheritance was an interesting feature to achieve some of the functionality required for the scenario. The integration of such distributed object inheritance into our model will be the core topic of this chapter. Another important identified concern was *strong code mobility*. We have not yet researched integration of proper object movement in our model. Supplying a `move` native as is done in Emerald (Jul et al., 1988) is found to be too low-level. We support the statement *move considered harmful* in (De Meuter, 2004). However, we will not pursue our quest for proper movement abstractions any further in this dissertation. One more language characteristic we will leave unexplored is *broadcast communication*, even though we find the idea of a *multivalue* (De Meuter et al., 2003a) very appealing to achieve this. A fourth concern was the presence of *autonomous processes*, which has been extensively dealt with in the previous chapter with the introduction of *active objects*.

Even though dPico does not address such important concerns as partial failure handling and mobility, essential for next-generation "ambient programming languages", the distribution model is still powerful enough to express a variety of interesting problems in a natural way. We have opted to elaborate only part of the required features to ensure that they integrate well within one language, rather than just designing independent and unrelated features that may work well on example programs but which cooperate badly. Such features tend to interact in unexpected

ways with other language constructs if they do not fit well into the existing language's design. After all, true language engineering is based on consolidation and a well-considered choice among alternatives to achieve the desired behaviour (Hoare, 1973).

6.1.1 Targeted Applications

Hoare (1973) has stated that it is very important for a language designer to envision the goals of his programming language. In other words, we must think about the target software applications of the language under construction. Since we do not fulfill all of the requirements that have been identified in the scenario in chapter 1 it is obvious that dPico will not be used to program Maria's P-Com. However, our moderate language could be used to develop *"connected applets"* (De Meuter, 2004). In such a software application, the user requests and "downloads" a piece of runnable code or an object that will remain connected to the server that has created it. The link between the "applet" client and the spawning server is a privileged parent link, which allows the client to access functionality that is not necessarily part of the public interface of the server.

In the scenario of chapter 1 the control object that is delivered from the meeting room projector to Maria's P-Com can be seen as such a connected applet. It is an object consisting of both data and executable code. Using this control object, Maria can then for example adjust settings, and project images. This functionality is not directly available on the projector itself. It can only return control objects to users who have the correct access rights.

Security is not the only advantage of this mechanism. Imagine a simple client-server chat application. The chat server can spawn "chat applets" whenever some client is requesting to collaborate in the chat session. The server can respond to such requests by returning the chat applet which remains connected to the server through its parent link. The server will keep track of where the chat applet is located, so that it can route messages to it from other clients. An important feature required here is that a parent can get a reference to its created child! Due to encapsulated inheritance, this is possible in dPico, and the server can keep track of his children. Because a child can always delegate to its parent, the server is an ideal mediator to ensure communication between two separate applets not having a direct communication link. We have written such a software application in dPico. The source code of this example program illustrates the more important features of the language and can be found in appendix B.2.

6.1.2 Extending The Concurrency Model

It is natural to build the distribution model on top of our concurrency model designed to cope with concurrency issues. We have extensively motivated that in our particular context, distribution and concurrency are tightly coupled. Nevertheless, such a choice needs some consideration. It is to be expected that cPico will

need adjustments to cope with the complex problems posed by distribution. Before explaining the required changes, we will first review and motivate why we feel that the concurrency model we have presented is a good starting point for crafting dPico.

Let us first briefly summarize cPico's main arsenal of language features to tackle concurrency issues. We have introduced *active objects*, equipped with buffers to ensure serialized execution of requests. Method invocation on *serialized objects* ensured *atomic method invocation*. Finally messages sent to active objects were handled *asynchronously*, to increase their autonomy. In order for the sender of such messages to be able to retrieve results and to synchronize on the call, *transparent promises* were introduced. This proved to be a key feature of cPico since it combines the high degree of concurrency that can be obtained from asynchronous message passing, and the simplicity that comes from a controllable way of thinking promoted by synchronous message passing.

We need to maximally exploit these features when transcending to the realm of distribution. As we will see, active objects will be the hallmark of our distribution model, since we can reuse many of cPico's concepts for active objects, irrespective of the active object's *location*. In a distributed setting, we value the power of their autonomy and their asynchronous communication.

In the previous section we have introduced "connected applets" (De Meuter, 2004) as client objects that have a distributed parent or delegation link. To this end, we need to introduce the concept of a "netview". A netview is in essence a view that is created on an object that resides on another node in the network. Active objects seem an ideal candidate to express the client and server objects (the child and parent in a netview relation). Both have their own thread of computation, so they can autonomously process requests and each execute code at a different location. In order to be able to express such delegation relation between active objects, some changes to the inheritance scheme of cPico are required. Recall that cPico only allows for delegation relationships between passive objects. The necessary changes will be documented in section 6.3.

Notice that the goal of our distribution model is to achieve transparency of location. This transparency is used to abstract away from the underlying "processor cloud" in which the programs live. Since there will be a lot of asynchronous communication between different active objects, the dPico programmer will have to be wary of synchronization issues. Therefore, we cannot simply make the distinction between active and passive objects entirely transparent. We will see how a clear-cut distinction is upheld between active and passive objects by re-organizing them into separate delegation hierarchies. This distinct organization is crucial for the programmer to recall which objects are active and thus require special attention when it comes to synchronization.

6.1.3 Object Extensions Revisited

Before turning to the distributed aspects of dPico, its object creation and extension mechanisms are explained. These mechanisms differ somewhat from the approach taken in Pic% and cPico. A new extension mechanism is incorporated in dPico is because there is a strong need for more *control* over object hierarchies. cPico's extension mechanisms using `capture` is too liberal. Moreover, such scheme provides no "static" information to the interpreter to discriminate between constructor functions and ordinary functions. In a distributed context, such static information will prove to be crucial. Before introducing dPico's distributed concepts, a more controllable object extension mechanism is discussed.

Recall from section 2.5.3.2 that Pic% objects can be extended by "capturing" a dictionary that is extended during function application with a call frame. The `capture()` native simply returns this extended dictionary, and is as such sufficient to create new objects. We have also introduced a `clone` native which could clone its receiver up to a given dictionary. Later on, in chapter 5, we had to adapt the semantics of `capture()` in cPico, since mere "call frames" were no longer full-fledged objects (see section 5.7.2). The `capture` native had the additional task of "converting" call frames into objects. Yet, the way in which objects were created didn't change conceptually. For instance, to create a point object with two accessor methods, it suffices to write (in either Pic% or cPico):

```
point(x,y) :: {
   getx() :: x;
   gety() :: y;
   capture()
}
```

We have already addressed some problems using such an object creation scheme in the context of *cloning* objects in section 5.4.1.1. There, we have advocated the use of a scope function termed `cloning` to create implicit clones of objects while allowing arbitrary code to be executed *in the context of the clone*. In this section, we will further extend this approach to also include plain object extensions (views) and imperative mixin methods. At first, the benefits may seem a bit superficial, but as we will see in section 6.4.1, this new object creation mechanism will prove to have more fundamental advantages in the context of distribution. Also, although the new method of object creation might at first seem awkward, we will present a syntactic extension to Pic% in section 6.1.3.4 which will greatly ease the use of our new concepts.

The approach to object extension we will take here closely resembles the one taken in Agora (De Meuter, 1998). As explained in section 2.5.2.2, Agora introduces special `VIEW:` and `CLONING:` reifiers, which install methods whose body is executed in the context of new objects upon invocation.

dPico introduces three new native functions – `view`, `mixin` and `cloning` –

which will achieve similar semantics[1]. Each of these new native functions takes exactly one argument expression, which must evaluate to a function or a closure. Its return value is a behaviour-modifying "wrapper" around the function. Invoking such a wrapper will apply the underlying function, but in a new or modified object context. Subsequent sections will explain each of the natives in more detail.

6.1.3.1 View Methods

The `view` native returns, for any function or closure, a "view version" (or "view method"). When applied, the view method's body will be evaluated in the context of a view extending the current `this()` object. Thus, evaluating `view(fun)` is equivalent to evaluating `this().view(fun)`. Since the view function is evaluated in the context of the view, references to `this()` in its body will point to the *extension* itself, while `super()` will evaluate to the object under extension. This is in contrast with the `capture`-approach, where `this()` always points to the object under extension. Figure 6.1 illustrates the difference between both approaches using the simple example of a `mammal` object being extended to a `human` object. Hence, `human` is a view on `mammal`.

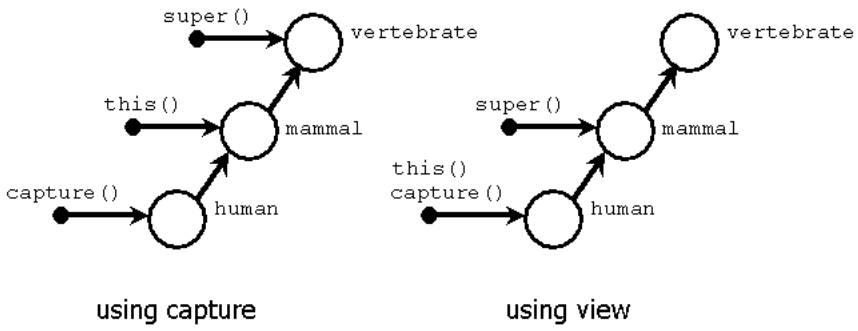

Figure 6.1: Object extension using `capture` versus using `view`

The point object is defined using view functions as follows:

```
point :: view(
  lambda(x,y) :: {
     getx() :: x;
     gety() :: y
  })
```

Notice that point is no longer declared as a function, but rather as a plain reference, which will get bound to the result of applying `view` on some function named `lambda`. Remark that this is not an anonymous function as in Scheme. It is merely

[1]This `cloning` method replaces the scope function approach of cPico, explained in section 5.4.1.1.

a function named `lambda`. Notice also that there is no `capture()` or `this()` expression necessary to "return the object". When a view method is applied, it will discard the result of its underlying function and return the view itself. If we were to use `this()` inside the body of `lambda`, it would refer to the view itself.

The attentive reader may have noticed that `capture()` refers to the view frame itself during the execution of a view method. This is because view methods, when applied, do not create an extra call frame to bind their arguments. Instead, arguments are bound in the object extension directly. This allows for the x and y parameters to become instance variables of the point, just like this was the case in the `capture()` approach of Pic%. Using the `view` method approach, the `capture()` native has become obsolete and will no longer be used as the object constructor.

6.1.3.2 Mixin Methods

Mixin methods exhibit more complex behaviour, since they allow for an object to be imperatively *changed*. A mixin method, like a view method, executes its underlying function in the context of the mixin itself. As is also the case with view methods, mixin methods always implicitly operate on `this()` if no receiver is specified. When a mixin method is invoked, a clone of the receiver is created. The parent of the receiver is subsequently assigned to this clone. Next, the receiver object is *cleared*: all of its variables and constants are removed. The mixin method's body will then execute in the context of this emptied object. Figure 6.2 clarifies the operation of `mixin` through an example. In the example, a 3D point, being an extension of a 2D point is imperatively changed into a 3D sphere.

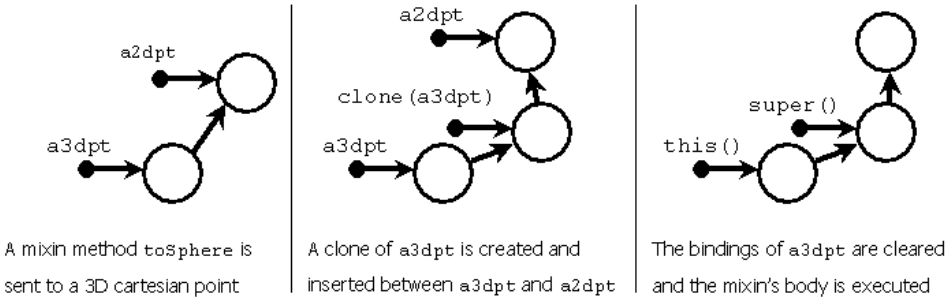

Figure 6.2: Applying a mixin method to an object

The third figure shows that in the context of execution of a mixin method, `this()` still refers to the original receiver (the former 3D point, being turned into a sphere), while `super()` now denotes the clone of the receiver's *old* behaviour (being a 3D point). As one can see, arguments and definitions in the mixin's body will be bound in the original receiver. Thus, any object having a reference to the 3D point will now observe the point as a sphere. Using super sends, the sphere can still access

its old point behaviour. When applied, mixin methods will always implicitly return `this()`, that is, the "extension".

6.1.3.3 Cloning Methods

The last of the three special methods are the `cloning` methods. Cloning methods solve the problems discussed in section 5.4.1.1 relating to the problem of passing arguments to the `cloning` scope function. There, a rather "ugly" solution was taken by executing the `cloning` scope function in a copy of the call frame, such that variables defined at "call time" became visible. As already mentioned at that point in the text, dedicated cloning methods alleviate this problem, since they can take arguments just like normal methods.

A cloning method executes its body in a *clone* of the receiver and always returns this clone. Function calls of cloning methods operate on `this()`. Unlike view and mixin methods, cloning methods *do* create a call frame, meaning that definitions made in the cloning method will no longer be visible inside the clone when the actual method returns. Cloning methods have the advantage that inside their body, `this()` already refers to the clone, and variable initialization can take place. Figure 6.3 illustrates the evaluation context during cloning method application.

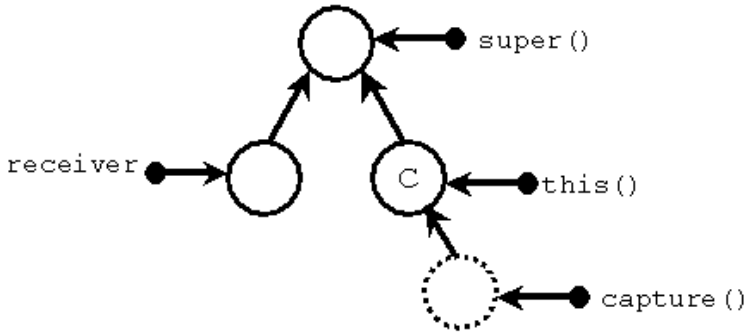

Figure 6.3: Applying a cloning method to an object

To illustrate the usage of cloning methods, we will show how to clone a point object:

```
point :: view(
  lambda(x,y) :: {
     getx() :: x;
     gety() :: y;
     clonePoint :: cloning (
       lambda2(nx, ny) :: {
         x := nx;
         y := ny
```

```
    })
})
```

6.1.3.4 Extending Pic%'s Syntax

As can be witnessed from the examples in the previous sections, creating view or cloning methods using the new concepts is rather cumbersome due to the absence of anonymous functions in Pic%. Therefore, we have added some syntax extensions to Pic% which simplify the creation of special methods. The main problem is that we had to introduce temporary functions merely to serve as input for `view`, `mixin` or `cloning`, because Pic% lacks anonymous functions. These temporary functions merely pollute the namespace and should be avoided.

To be more precise, we did not introduce new syntax, but have rather defined semantics for syntax previously "not in use". In Pic%, it is not possible to "define a message send". That is, the Pic% interpreter will throw an error when trying to evaluate `o.m(args):body`. The left-hand side of a definition, declaration or assignment should always be a plain invocation, not a message or super send. We will allow such a syntax to enable the use of "anonymous functions".

When evaluating `exp.m(args):body`, the expression `exp` will first be evaluated to a function f. If the result of evaluating `exp` is not a function, an error is raised. The function f will be immediately *applied* to a new function comprised of the name `m`, the arguments `args` and the body `body`. Yet, the method `m` will *not yet be bound in the current dictionary*. Rather, `m` will eventually get bound to the result of applying f to the unbound function. We can approach the behaviour of these semantics by writing:

$$f.m(args) : body \equiv m : f(m(args) :' body)$$

Where $:'$ is some imaginary definition operator which *creates* a function but does not bind it in the dictionary. Put differently, $:'$ can be used to create "anonymous lambda functions" (although methods in Pic% still always carry a name). For `f.m(args)::body`, the semantics stay largely the same, but instead the result of applying f to the function is now bound to a *constant* m instead of a variable.

How can our new object creation schemes outlined above benefit from this new syntax? When the native `view` is used to play the role of f, "view methods" can now easily be created by simply annotating a normal Pic% function with a `view` qualification. The same can be done for creating mixin or cloning methods. We will illustrate the new syntax on our previous point example:

```
view.point(x,y) :: {
     getx() :: x;
     gety() :: y;
     cloning.clonePoint(nx, ny) :: {
        x := nx;
```

```
        y := ny
     }
}
```

When evaluating `view.point(x,y)::body`, `view` will be evaluated to the native function $view$, which is subsequently applied to the function `point(x,y)::body`. The result (a "view method") will be bound to m, instead of the function that would result from the same syntax *without* the view qualification. The same goes for mixin and cloning methods. We believe this new syntax allows for an expressive creation of "wrapper methods", while keeping the number of concepts in the language minimal. Also, no "real" syntax extensions to Pic% were necessary. In fact, the parser did not even have to be adapted to parse the above syntax.

Notice also that the newly defined syntax is really completely independent from these three new natives. The syntax can be used for other abstractions as well. Consider the following example which "lifts" methods to become *critical sections*.

```
critical(semaphore) ::
    lift(function) ::
       lifted@args :: {
          semaphore.P();
          function@args;
          semaphore.V()
       }

view.bankAccount(bal) :: {
   mutex: makeSemaphore();
   critical(mutex).withdraw(amnt) :: bal := bal - amnt;
   critical(mutex).deposit(amnt) :: bal := bal + amnt;
}
```

Usually, functions which will be used at the left-hand side of a "message definition" are higher-order functions that return another function. Thus, they can be regarded as "function transformations", although this does not necessarily have to be the case. In the example above, `critical` is a function taking a semaphore object as an argument. It returns a temporary function, which is the function that will be applied to m. This function in turn returns an extended function which "wraps" the original function in semaphore wait and signal message sends. This allows for methods to be easily flagged as "critical sections" on a given semaphore, as is illustrated by the simple bank account whose `withdraw` and `deposit` methods will be mutually exclusive.

When defining a language, it is important to define complete or consistent semantics. In Pic%, for example, one can always define, declare or assign any of the three invocations (reference, application and tabulation). Up until here, we have

merely described semantics for defining or declaring a qualification consisting of a receiver and an application. For completeness, we have also extended Pic% with the proper semantics to deal with the definition or declaration of qualifications containing a reference (`f.x:exp`) and a tabulation (`f.t[i]:exp`). Using our previous notation, their intuitive semantics are:

$$f.x : exp \equiv x : f(exp)$$
$$f.t[i] : exp \equiv t : f(t[i] :' exp)$$

For references, the syntax does not really offer any advantage (as can be witnessed by the lack of the $:'$ operator). For tabulations however, it allows a function f to intercept a table before binding it to the environment. More detailed semantics of message definitions can be found in appendix A.4.1.

Concluding, it can be seen that dPico's extension mechanism through "wrapper" functions provides for more controlled object extensions. We have also claimed an amount of "static" information was needed to discriminate between constructor functions and ordinary functions. The above object creation scheme effectively allows the interpreter to discriminate between "view methods" and normal methods *before even applying the function*. The advantages such information brings will become clear when discussing active object extensions in section 6.4.1. In the following sections, dPico's distributed aspects will be covered, starting with the parameter passing mechanism for message sends to remote objects.

6.2 Distributed Parameter Passing Semantics

As has been extensively discussed in chapter 4, an important part of distributed programming languages is their definition of proper parameter passing semantics when messages are sent across the boundaries of the local address space. This section will discuss how object references are dealt with when objects are passed as arguments during a remote method invocation.

Our main approach is to base our distribution model as much as possible on our concurrency model. Since the latter is mostly built upon active objects, we will also consider *active objects as the unit of sharing* in a distributed setting. That is, we will allow *only* active objects to be shared between multiple Pic% interpreters. Although this approach might seem a bit restrictive at first, it simplifies the semantics of the language a great deal. First, let us argue why sharing only active objects is advantageous in our context:

- Recall that active objects are equipped with their own message queue and with a serialized behaviour. As such, they find themselves naturally encapsulated and protected from race conditions. The buffer ensures that only one method is executed at a time. Also, the message queue can automatically buffer incoming network requests, just like it can queue up local message

requests. Thus, there is no need for extra queues to buffer network requests to passive objects.

- Allowing only remote references to active objects ensures that any call to a remote object will *always* be asynchronous. This is because the only remote objects are active objects, and as was already mentioned in section 5.5, any call to an active object is always asynchronous.

- Since remote method invocations will always be asynchronous (as they must be method invocations on active objects), we gain a huge amount of efficiency. In a distributed context, asynchronous method invocation is highly advisable as it allows for performing a lot of communication without blocking the sender. Blocking the sender during a remote method invocation is inefficient since such invocations are inherently slower than their local counterparts. Idle times could quickly become a performance bottleneck for communication-intensive objects. The main disadvantages of asynchronous method invocations – the lack of return value and corresponding caller-callee synchronization – are countered by our usage of promises.

- As will be discussed in section 6.2.2, allowing only active objects to be remotely referenced eases implementation-level object management and reduces the number of remote object references if passive objects are involved.

- Allowing only active objects to be shared simplifies the language's semantics by allowing for a simple set of programming guidelines. For example, if the programmer wants to share data across the network, he is forced to *encapsulate* that data in an active object. Therefore, he is forced to think about the concurrency issues involved, and access will be guaranteed to be serialized.

Of course, disallowing remote references to simple passive objects has its downsides too. In subsequent sections, we will clarify how our restriction on sharing is enforced throughout the language. Where appropriate, we will also discuss the disadvantages of the imposed restrictions.

6.2.1 Publishing Objects

When trying to make two Pic% interpreters communicate, the first issue is how to make the interpreters able to access each other's objects. This "service discovery" mechanism will be discussed in detail in section 6.6. For now, it is only important to know that we allow *only* active objects to freely "publish" themselves. That is, only active objects will be directly accessible by objects residing on other interpreters. Allowing passive objects to publish themselves would immediately break our constraint on remote references.

6.2.2 Parameter Passing and Return Values

As mentioned in section 4.2.5.2, having the ability to send messages across a network requires one to properly define how arguments are passed on to a remote method. This is necessary when invoking methods on remote objects. Since only active objects are allowed to be referenced remotely, we will be using very simple parameter passing semantics for both arguments and return values for remote method invocations:

- Active objects are *always* passed by reference.
- Any other Pic% value is *always* passed by copy[2]. Note that this rule is also applicable for *passive* objects.

Next to being directly "published", an active object can thus also be remotely referenced by passing it as an argument to a remote method. Passive objects present problems, however. They cannot be passed by reference, since this would force us to provide remote references to passive objects. The alternative is to pass them by *copy*, which means that a copy of the object is passed to the callee.

One problem that immediately manifests itself is *how much* to copy and send on to the receiver. Our semantics are to always *copy the entire transitive closure of the object*. Two remarks are in place here. First, copying the entire transitive closure is the only viable solution to uphold the constraint on remote references. If only the parameter object would be copied, and this object has references to other passive objects, these references would have to be "cut off" and replaced by remote references to passive objects. Second, note that this transitive closure is only taken up to the point where an active object is encountered. That is, active objects within the transitive closure of a passive object are still passed by reference. Thus, active objects can be seen as nodes where the object graph being transmitted is implicitly pruned.

The policy of copying the transitive closure of a passive object has both advantages and disadvantages. The obvious downside is expensiveness. Copying a cyclic data structure incurs some overhead, making parameter passing more expensive than plainly passing a simple remote reference. Second, the remote method will receive a copy of the argument, meaning that any changes made to such a passive object are *not* reflected in the original object. There is no replication management scheme to keep the copies consistent. This enforces the use of active objects to accomplish network-wide sharing.

The use of the copy semantics has some advantages as well. First of all, since less objects are passed by reference, there is less implementation-level object management to be performed. There will be less remote references, allowing for smaller "remote object tables", which manage the mapping between local and

[2]Unique values like `void` will remain unique per virtual machine, however.

remote references. Also, execution of the remote method itself can be sped up because method invocations to passive arguments can be handled locally. They do not have to travel back over the network to the caller.

Our approach of disallowing sharing of passive objects between multiple interpreters can also be observed in a local, concurrent setting in the language Eiffel// (Caromel, 1990, 1993) in which passive object arguments are copied when invoking a method on a local active object (Ehmety et al., 1998). This completely eliminates shared passive objects between multiple "processes" or active objects, regardless of locality. The advantage is that passive objects will never be prone to race conditions. On the other hand, such semantics are expensive. We allow passive objects to be shared by local active objects freely. For this purpose, we have provided serialized objects as a protection mechanism against the race conditions. The different semantics for parameter passing to local or remote active objects are not without consequences. Section 6.11 will get back at this.

6.2.3 Passing Delegation Links

We have noted that when passive objects are passed by copy, a transitive closure of the target object is always copied, stopping at active objects which are replaced by remote references. It is important to consider what will happen to the *parent* of an object. In other words, is the transitive closure also taken across delegation or parent links? Since parent links always refer to passive objects in the inherited cPico object model, the parent *must* be passed by copy too, to prevent any remote reference to passive objects. This uncovers a major weakness in the simple semantics of disallowing shared passive objects in combination with cPico's object model: parent sharing across the network is *lost*! A parent will always be copied whenever one of its children is passed to another interpreter. The original parent will not be able to support sharing of state across the network. cPico's object model will thus have to be refined if dPico is to be rendered useful.

Recall from chapter 1 that we envisaged distributed object inheritance hierarchies to be a prime feature of the language under study. Therefore, in section 6.3 distributed parent sharing will be reintroduced *without* sacrificing the simple parameter passing rule introduced in this section. The answer lies in the creation of delegation links to active objects. Such delegation schemes have a price to pay in increased complexity, however. Therefore, we have evaluated some viable delegation alternatives with regard to semantic simplicity and integration in the existing model. The latter is important to ensure that the new solution does not break with the concurrency model introduced in cPico.

6.3 Implications for Parent Sharing

The hypothetical model we have introduced in the previous section has one major drawback. It prevents distributed parent sharing, due to the fact that all passive

objects are passed by copy across the network. In cPico, all delegation links refer to passive objects and delegation to an active object is rerouted to its passive behaviour. Using the outlined parameter passing semantics, we would always copy the entire passive hierarchy when making a netview (i.e. a view on an object that resides on another location). Since we can only pass around active objects the solution must be to "activate" the hierarchy. Since active objects are passed by reference, we can only have a distributed object graph if we have pointers to active objects.

There are two different ways to introduce active objects in the hierarchy, which we will both illustrate using a simple (albeit abstract) example. Consider two active objects `aParent` and `aChild`, where `aChild` should be seen as an active extension of `aParent`. In the hypothetical model we have presented in the previous section, this relationship is implicit, since they are only related due to the fact that the behaviour of `aChild` delegates to the behaviour of `aParent`.

In this section we will try to reorganize how delegation pointers are maintained between objects, to make the relation between `aParent` and `aChild` explicit. This means that the active object `aParent` appears somewhere in the delegation chain of `aChild`. Only then `aChild` and `aParent` can reside on different machines, since remote references can only be made to active objects.

6.3.1 Mixed Inheritance

A first solution might be just to allow the parent pointer of a passive object to refer to an active object directly. This evolution is shown in figure 6.4. Objects marked with a "thread symbol" are active objects. The object marked with an N represents the dictionary containing all native functions. This approach reintroduces a mixed inheritance scheme where both active and passive objects may appear. This reintroduces the awkward semantic peculiarities we have associated with it in section 5.5 of the previous chapter.

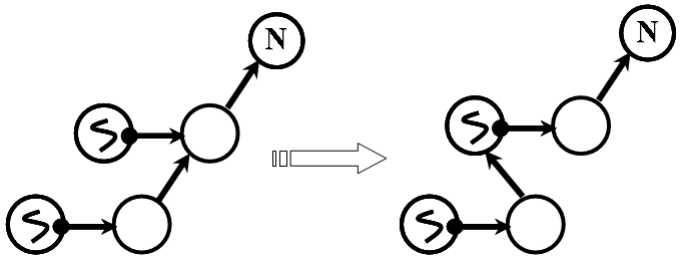

Figure 6.4: Restructuring Active Parent-Child relations with mixed inheritance

It is also important to remember that we do not want to change the rules for invocation we have proposed in chapter 5. We will repeat these rules here for the sake of readability. Messages sent to a passive object are always synchronous, whereas messages sent to an active object is always asynchronous. Delegation is always synchronous. A message is considered to be "delegated" when

1. it is sent to a child but implemented (found) in a parent,

2. when a super send is used to delegate the message explicitly.

In this mixed inheritance scheme keeping delegation synchronous may be difficult to uphold. The first case of delegation can be solved by immediately scheduling the request in the right queue[3]. However, keeping super sends synchronous, even if the parent is active (and can thus be remote) would mean that at least some invocations over the network are performed synchronously, which is a situation we want to avoid in our context. This is one of the reasons to refrain from using remote references to passive objects, so the mixed inheritance solution does not really solve the problem gracefully in all cases.

To cope with the restriction on remote references we would have to put severe restrictions on our language and prohibit an invocation `m()` if the method `m` is defined in an active parent object. This would also imply that an invocation of the form `.m()` is entirely prohibited if the parent would be active, since this would always force synchronous invocation on active objects. These are draconian measures to restrict method invocation without a simple and concise set of rules behind them.

Another problem with the mixed inheritance scheme is that the natives dictionary is only found at the top. Thus, in a distributed system where some delegation links cross machine boundaries, method lookup for natives may involve remote lookup. This is of course unacceptable, since even Pic%'s "control structures" like `begin` and `if` are natives. If we would want to keep the language realistic, this would require us to short-circuit the lookup of native functions. This would mean that we always first check whether an identifier is a native, and if so return this native before starting general method lookup. This would prevent native overriding, which shows that the mixed inheritance scheme really conflicts with existing language concepts.

We conclude that although mixed inheritance does allow for distributed inheritance, it also introduces a plethora of problems such as exceptional cases for method lookup and natives. These problems seem to suggest that a different, simpler delegation scheme is needed that still allows for distributed inheritance.

6.3.2 Parallel Inheritance

In cPico, we have advocated the distinction between active objects and their passive behaviours. However, the fact that distributed hierarchies are required – something which can only be done using active objects – seems to suggest that active objects require their own *separate* hierarchy. To support active object hierarchies, we will equip active objects with *their own* parent pointers whereas previously only the behaviour had pointers. The parent pointers of an active object can *only* point to

[3]With this approach we can schedule the request in the queue of the last active object we encountered.

active objects. Thus, the link between two active objects is now made explicit by connecting them directly, as is shown in figure 6.5. Note that, even though the figure depicts two objects marked N, in reality, only *one* natives dictionary is present on each virtual machine.

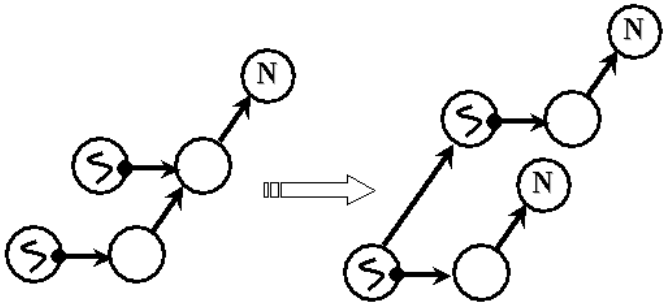

Figure 6.5: Restructuring Active Parent-Child relations with parallel inheritance

We have called this inheritance scheme *parallel hierarchies* since it is possible to maintain a parallel set of delegation hierarchies. On the one hand there is an active hierarchy composed solely of active objects, which may be distributed. On the other hand, there is a passive hierarchy comprised completely of passive objects. Such a passive hierarchy is local to an active object. The behaviour of the active object is in a sense the "root" of the local passive hierarchy.

This double hierarchy solves the problems we encountered with the application of natives with the mixed inheritance scheme in which natives could only be found at the top of the hierarchy. In this proposal with double hierarchies, natives appear locally at the top of *every* passive hierarchy, which ensures that natives will never be looked up remotely if they are applied to a passive object. If a native is applied on an *active* object, e.g. `ao.+`, such lookup requests *can* travel over the network, but this is of course needed to ensure that native overriding remains possible.

The parallelism introduced by our new inheritance scheme is also reflected in the evaluation context, in which the evaluator keeps track of such information as where `this()` and `super()` point to. In Pic%, an evaluation context was always fully identified by the triple (`current`,`this`,`super`). Due to the introduction of the active hierarchy and *multiple* passive hierarchies, this triple is not sufficient anymore. We also need to maintain active counterparts for each of these context parameters.

In cPico, the context already contained an `activethis` parameter, which is the active object currently executing the method. dPico contexts introduce three new context parameters. During method application, `acurrent` is the active object that has implemented the running method (i.e. it is the active object in which the method was lexically found). `asuper` is the parent of `acurrent`[4], pointing

[4]Since `asuper` is *always* the parent of `acurrent`, we have in our implementation only kept track of `acurrent`.

to the lexically scoped active parent. Finally, `athis` represents the active object that initially received the executing method. In this more complex active object delegation scheme, cPico's `activethis` should be interpreted as `acurrent` and is thus obsolete.

By restricting receiverless invocations like `m()` and `.m()` to the passive hierarchy, we can avoid the problems encountered with mixed inheritance. This is because we interpret such expressions in the context of the passive hierarchy only. Thus, `super()` always refers to a passive object and delegation is guaranteed to be synchronous. This is possible because of the clean separation between the two hierarchies. Figure 6.6 stresses this separation and is meant to provide a *mental image* of how dPico objects are structured.

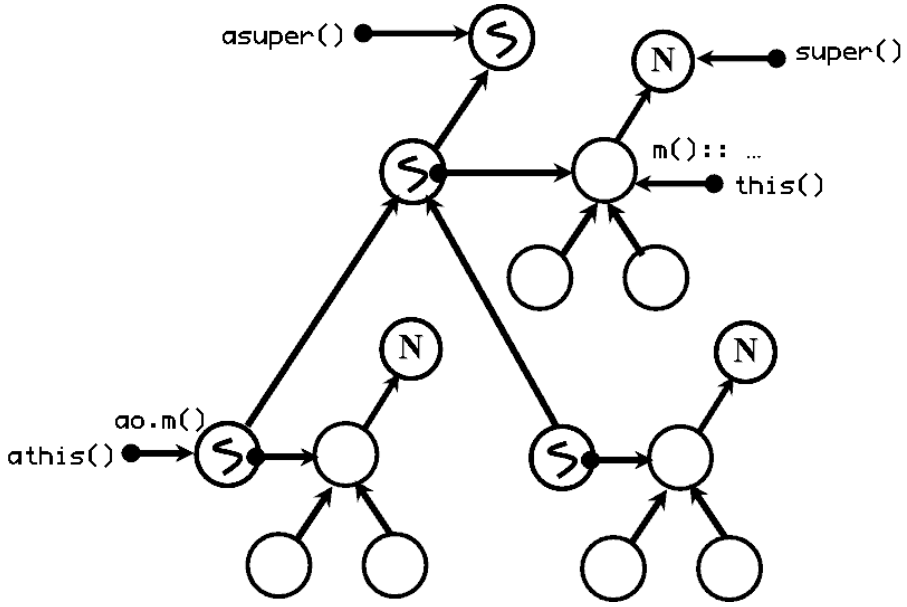

Figure 6.6: Identifying context parameters in parallel hierarchies

Another important observation is that `this()` is not necessarily the behaviour of `athis`. This will only be the case if `athis` equals `acurrent`. To see why, note that we do not want `this()` or `super()` to cross the boundaries of the current passive hierarchy. Therefore, `this()` is only late-bound with respect to the passive hierarchy, guaranteeing that a message sent to `this` cannot escape the passive hierarchy.

Of course sometimes a message may *want* to be able to escape from the current passive hierarchy. When evaluating `m()`, the method call is restricted to the passive hierarchy. If a message is to be sent to another passive hierarchy, `athis().m()` or `asuper().m()` can be used. Notice that crossing the boundaries of the passive hierarchy involves asynchronicity. Thus, the programmer must be cautious with the return value (to avoid deadlocks). However, crossing passive hierarchies via `athis` or `asuper` has the advantage that it is done through an explicit mes-

sage send to an active object. This should be a clear reminder to the programmer that such message sends are handled asynchronously and return a promise. The mixed inheritance scheme did not have such clear guidelines.

This scheme, using `athis().m()` for dynamic lookup of a method and `asuper().m()` for methods that are located in an active parent, is better suited towards a support for distributed object inheritance. This due to asynchronicity of all message passing occurring along the active inheritance chain. This is very useful since it is exactly this kind of communication that we want to establish across a network. The synchronous communication is limited to the passive hierarchy, which is always local due to the *call-by-copy* semantics for passive objects we have introduced in section 6.2.

One of the main reasons that the parallel delegation strategy will work is the fact that we propose a controlled extension mechanism for both passive and active extensions. This new mechanism replaces the `activate()` native which could at all times be used to make an isolated active object out of an existing passive object. In our current system where both passive and active objects are always structurally related, such an (active) object creation mechanism is too liberal. In the next section the constructs that have been introduced to achieve controlled extension of active objects are discussed.

6.3.3 Summary

In this section we have discussed two different mechanisms to introduce active objects as part of the delegation chain. This is essential in order to have distributed object inheritance, with sharing of distributed parents, a key feature of dPico. In order to be able to distribute a hierarchy, without copying objects, parental active objects are a necessity since they are the only values that are passed by reference.

The first mechanism removed the restriction imposed in cPico that passive objects can only delegate to other passive objects. Apart from reintroducing the problems we have discussed in chapter 5, a new range of problems arise when this mixed inheritance mechanism is employed in a distributed setting. The second mechanism, which is the one that is actually used in dPico, parallels the passive hierarchy with a new active hierarchy. In order to have more control over how hierarchies are built, we have abandoned the `activate` native in favour of a more rigid extension mechanism. For passive objects, these extensions have already been explained in section 6.1.3. The extension mechanisms for the active hierarchy will be explained in the next section. The semantics of method lookup on an active object in such a hierarchy, will also be described later on in section 6.5.1.

In order to support the parallel delegation structure a new set of natives were added that allow for asynchronous message sends to the active hierarchy. For clarity, they are summarized in table 6.1. All four natives are explained in the context of a message send to both an active and a passive object. The “owner” of a passive object denotes the active object in control of the passive hierarchy where the object is located. For a full review of all important natives in dPico with a brief

description of their meaning we refer to appendix C.

	`active.m()` `m` implemented in `aParent`	`passive.m()` `m` implemented in `pParent`
`this()`	`aParent`'s behaviour	`passive`
`super()`	`this()`'s parent	`pParent`'s parent
`athis()`	`active`	Remains unchanged
`asuper()`	`aParent`'s parent	Parent of owner of `passive`

Table 6.1: Summary of Natives Supporting Parallel Hierarchies

6.4 Distributed Object Inheritance

In the introduction of this chapter we have already mentioned that distributed object inheritance is one of the key features of the distribution model. In order to support it, the inheritance model among objects has been refactored which has lead to a separate hierarchy of active objects, better suited for distribution aspects. To ensure that the dPico programmer cannot escape structuring his objects as for example shown in figure 6.6, we need a controlled and concise extension mechanism for active objects. We will build upon the controlled extension mechanism of passive objects using `view` methods. We will also look at how active *scope functions*, similar to the ones introduced in section 5.4.1, are fit into the parallel hierarchies.

6.4.1 Active Object Extensions

We have introduced the concept of parallel delegation hierarchies in the previous section, without much attention to how such hierarchies would be constructed. This section introduces the active counterparts of `view` and `mixin`.

6.4.1.1 Active Views

An active view is very similar to a regular, passive view. The difference is that an active view is applied to the active part of the hierarchy. An active view creates a new active object whose parent will refer to the active object on which the view was invoked.

Striving for simplicity and minimality, we were first tempted to reuse the `view` native for passive objects, introduced in section 6.1.3, to create extensions on active objects as well. At first sight, this would be an ideal decision since it unifies the notions of both passive and active object extensions. The unification could be conceived as follows: when a `view` method is invoked on an active object, it creates an active object extension, when it is called on a passive object it creates a passive view. This approach has its limitations, since the view itself lacks the necessary context, to determine whether it will be applied on the active or on the

passive part. Its code will have unpredictable effects, because its semantics will depend on the type of the receiver of the view.

If a view is written as described in section 6.1.3, it would always have to be taken into account that this view can be used both to create an active view and a passive view. This also means that the `super` and `this` natives would not be freely usable anymore. Figure 6.7 illustrates this problem for calls to `super`. If a view would be written, `super` cannot be used to access a method m which can be seen lexically, since if the view would be applied to an active object, leading to the creation of an active view, the method is located in the behaviour of another active object. This can be observed in part a of the figure. In the same vain, `asuper` is also unusable, since the view might also be passive, in which case `asuper` will immediately start looking one level higher up in the active hierarchy, skipping the desired method. This is illustrated by part b of the figure.

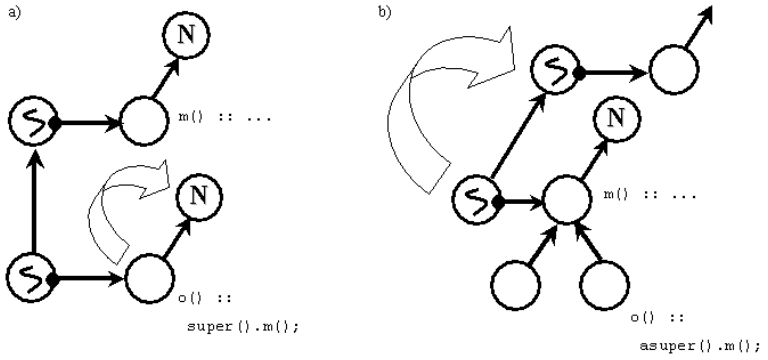

Figure 6.7: Super send problems with a unified view construct

Thus, view method bodies do not only depend on their specification but also on the kind of object they are dynamically applied to. This unpredictable context sensitivity is a bad property for a language concept. The examples further illustrate why we have ruled out the idea of unifying passive and active views. If we cannot perform such a unification, we must resort to the introduction of a separate active view. This is done in dPico using `aview`.

The `aview` native is in fact a function lifter that is quite similar to the `view` as it is defined in section 6.1.3. In the case of `aview`, a function is lifted as an "active view function". Just as `view` always creates an extension of its passive receiver, or `this()` if used receiverless, `aview` creates an extension of its active receiver, or `athis()` if none is present. Figure 6.8 illustrates the creation of an active view nested somewhere in the passive hierarchy. The newly created active view is denoted with with a dotted border. The context parameters shown are those active during the active view's body evaluation. We assume the active view function to have been applied to the object denoted by `asuper` in the figure.

Note that in this figure we clearly see that the passive context pointers are pointing to the passive object hierarchy of the active object that we are constructing.

This allows us to make parameters of an `aview` method to become variables in the behaviour of the new active view. Using this mechanism, we automatically lose track of the passive lexical environment in which the `aview` was defined, which can be clearly seen in the figure. This forces the programmer to pass everything he will need as a parameter since he cannot depend on "passive lexical visibility". We think this is a good practise since this context will not be available either when invoking messages on the active view. As such we introduce uniformity and make the programmer consider explicitly which references he may need in the newly created active view.

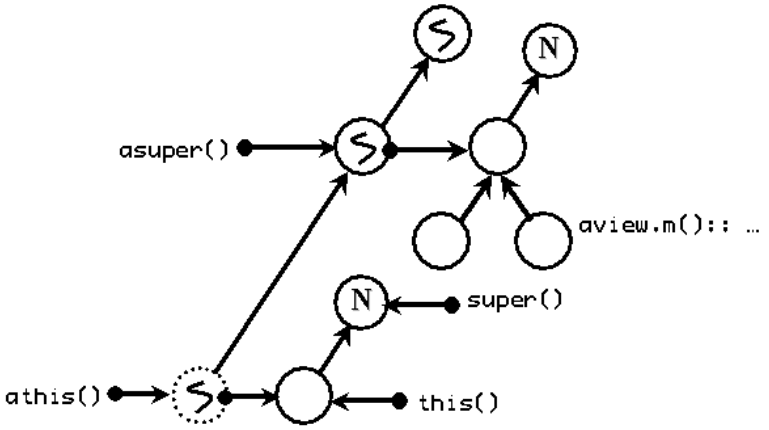

Figure 6.8: Active view creation with location of context parameters

The importance of active views lies in their natural support for distributed inheritance, in fact `aview` is *the only way to create a delegation link over the network* in dPico. Since we have not considered a move operator for active objects, we can only create a distributed hierarchy using such a netview, which is an active view on a remote active object. When an active view method is applied on an active object, the view is *always* created at the site of caller, even though the object under extension may be remote.

Here the real power of the function lifting approach to object extension is uncovered. As we have explained, wrapper functions such as `view` and `aview` allow us create a specialized function, altering the evaluation context of the original function prior to application. Imagine we still were to use `activate()` for creating active views. If we were then to send a message to a remote active object, the remote object would process the message, and finally execute the `activate()` native. This would create an active object extension at the location of the remote object. Since the parameter passing semantics we have introduced in section 6.2.2 clearly state that active objects are only passed by reference, the extension would be passed back as a remote reference. So, rather than having a distributed inheritance relation we would end up with a local inheritance relation on a remote location and a remote reference to the child.

This evaluation behaviour can be avoided by "annotating" functions at the implementation level. This is exactly what is done by the "wrapper functions". These annotations allow the interpreter to change its default evaluation behaviour with respect to function application. Function application in dPico, just like in Pic%, follows a recipe-query (Briot and Yonezawa, 1987) scheme, in which a method must always first be "downloaded" to be able to access its formal arguments. These formals are needed to ensure call-by-name parameter-passing semantics (there may be some actual arguments which cannot just be evaluated right away). Since functions can now be annotated as active view constructors, a remote interpreter invoking the method is alerted that the wrapped function is to be executed locally. Indeed, the "static information" provided by the wrapper gives the interpreter essential information *before* the method body is evaluated. The active view's body is then evaluated locally, creating an active view on the machine of the caller, which avoids having to implement a *move* mechanism for active objects. Inside the `aview` body the parent of the view, which may be located remotely, will be available as `asuper`. A concrete example that uses this extension mechanism is given in section 6.7.

6.4.1.2 Active Mixins

A second form of active object extension is the active mixin. The introduction of `amixin` next to `aview` makes the active object extensions symmetric to their passive counterparts. Just like active views mirror passive views by extending objects, active mixins mirror passive ones by imperatively *modifying* existing objects. Recall from section 6.1.3.2 that mixin methods always implicitly act upon `this()` when invoked receiverless. Likewise, active mixin methods always implicitly operate on `athis()` in such an invocation context.

Active mixin methods can be used to imperatively change a target active object. The net effect of applying an `amixin` method to a receiver is that the active object's behaviour is replaced by the new behaviour as specified in the amixin's body. The old behaviour will be placed in a clone of the receiver. Subsequently, the receiver's parent is set to this clone. As such, the old behaviour implementation is still accessible through sends to `asuper()`. To clarify the operation of active mixins, figure 6.9 depicts the creation of an active mixin, together with the context of evaluation during application of the amixin method.

6.4.2 Active Scope Functions

In section 5.4.1, we have introduced *scope functions* which execute their single argument expression in a specified scope. The two most important scope functions were `this(exp)` and `super(exp)`. Both were interesting since they allowed for flexible parent-child communication. Moreover, they provided for safe concurrent access to objects since they defined critical sections when `this()` or `super()` was serialized. In this section, their "active counterparts" are introduced: the active scope functions `athis(exp)` and `asuper(exp)`.

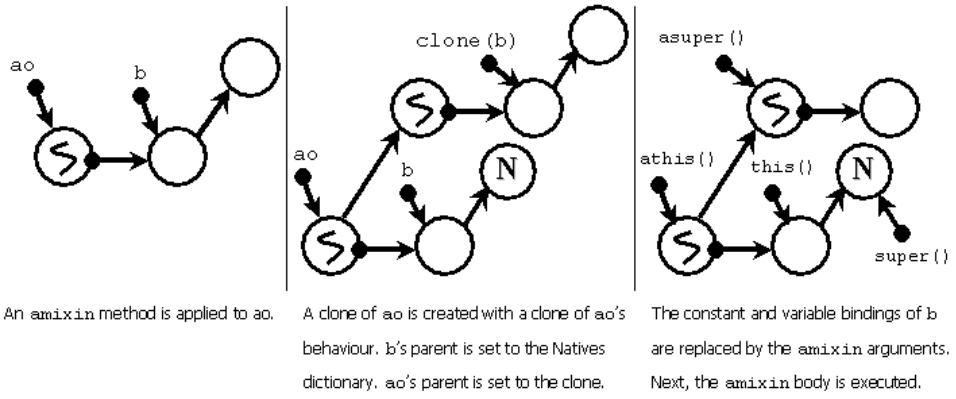

Figure 6.9: Active Mixin Method Application

The intuitive semantics of these active scope functions will probably be clear to the reader. `athis` evaluates its argument expression in the behaviour of the dynamic active receiver (`athis()`), while `asuper` evaluates hers in the parent of the lexical active object. The major distinction between the passive and the active scope functions is that whereas `this(exp)` and `super(exp)` are synchronous method invocations, `athis(exp)` and `asuper(exp)` follow the asynchronous tradition of active objects. This means that a call to an active scope function will return a *promise* that will be fulfilled once the call is complete. Active scope functions will thus be scheduled like any other asynchronous method invocation on an active object. `athis(exp)` will be evaluated by `athis()` itself, while `asuper(exp)` will be evaluated by `asuper()`. Notice that these semantics make sense, as we can interpret `athis(exp)` as `athis().eval(exp)`, which is a simple message send to an active object.

Recall that the `this` scope function preserves "late binding of self" by ensuring that `super(this())` is equal to `this()`. Analogously, `athis` preserves "late binding of active self", since `asuper(athis())` equals `athis()`. This allows for easy communication between an active child and its parent and vice versa. An expressive example of such communication and a good illustration of the use of scope functions is given in section 6.7.

There is one important remark to make regarding synchronization. The scheme of making active scope functions asynchronous in combination with late binding of active self can introduce a subtle deadlock if the programmer is not careful. If a certain active object evaluates `asuper(exp)` and subsequently blocks on the returned promise, he can only continue after his active parent has evaluated `exp`. However, if this expression somewhere contains a call to `athis` that will also block, the parent object will in turn wait for its child, leading to a simple cyclic wait – a deadly embrace.

Figure 6.10 illustrates such deadlock through the use of a simple example, calculating $x + y + z$, where x and z are defined in the child and y is defined in the

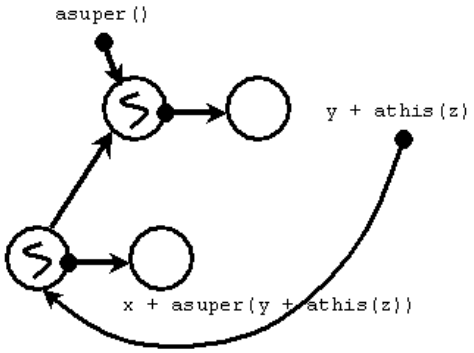

Figure 6.10: Deadlock using Active Scope Functions

parent. This is really a contrived example merely used to illustrate the deadlock. When the child invokes the `asuper` function, it will have to wait for the promise to be fulfilled to carry out the + operation. Likewise, the parent will wait for the child for similar reasons, and deadlock occurs.

We believe that by clearly distinguishing between active and passive scope functions, we allow for a better understanding of their semantics. The "a" prepended to each native regarding active objects should warn the programmer that he is dealing with asynchronicity and that he has to be careful when to touch the return value. Especially when sending messages to `athis` some care has to be taken. Evaluating `touch(athis().m())` will usually immediately deadlock an object which will be waiting for itself. However, due to late binding of active self, it is possible that `athis()` is a different object than the executing object, in which case the code fragment might not deadlock. In general, it is recommended that a self-send to `athis()` is only touched in the context of the `asuper` scope function, since we then can then at least guarantee that `athis()` will not be equal to `asuper()`.

6.5 Active Object Method Invocation

We still have to explain how a method invocation on an active object is handled in the distributed version of Pic%. Such method invocation is slightly more complicated because the receiver can either be a local or a remote active object. In the case of remote receivers, network calls will have to be performed to achieve method invocation. As a running example throughout this section, the invocation of `ao.m(arg)` will be handled where `ao` is assumed to be an active object, `m` denotes a valid method, and `arg` refers to any Pic% value. As for any Pic% method invocation, applying a method on an active object involves a two-step process:

Method lookup This means we will have to search for the implementation of `m`, starting the lookup in the receiver `ao` and possibly *delegating* to parent objects. As mentioned in section 5.8.2, if `m` is bound to a function, method

lookup will always return a *closure* containing the necessary context parameters to ensure static scope.

Method application Method application starts by binding the actual arguments to the formal arguments of the function wrapped in the closure in a *call frame*. Subsequently, the method body is evaluated in the proper context, consisting of the call frame and the necessary objects stored in the closure.

The following sections will go into some more detail on both aspects of method invocation on active objects.

6.5.1 Active Object Delegation

The purpose of active object delegation is to locate the method `m` in a chain of active objects. The lookup mechanism is reminiscent to that for passive objects, yet there are some notable differences. First of all, a lookup request is handled separately for local and remote objects. Local objects respond to a lookup request for a constant by searching for a given identifier in the constant part of their behaviour. If the method is not found there, the lookup request is synchronously *delegated* to the active object's parent. This is possible since active objects now have their own parent, apart from the parent of their behaviour. If the topmost active object (which is normally the *main* active object) cannot find the requested identifier, lookup ends in the natives dictionary.

If we encounter the requested identifier along the delegation chain, and it is bound to a function, the function is wrapped in an *active closure*. Such an active closure stores the function together with two active object context parameters: the "current active object" `ao'` denoting the active object in which the method was found, and the "receiver active object", which is the original dynamic receiver of the message (`ao` in our example). Notice that such an active closure will never contain passive objects, which is very important since this closure can travel across the network. This can happen if `ao` (the receiver of m) and `ao'` (the container of m) reside on different virtual machines. When encountering a remote active object in the delegation chain, a network request for the lookup of m is sent to it. The reply to this network request is either an active closure if m represents a function or simply the value of m otherwise. Active closures have the same goal as their passive counterparts: writing simply `ao.m` instead of `ao.m(arg)` must evaluate to some closure that captures necessary context information to be able to correctly apply m later on. Hence, dPico features *first-class remote methods*.

Notice that lookup requests initiated in active objects will *only* consider identifiers in their *direct* behaviour part, not in any parent of their behaviour. To see why, notice that if passive delegation would be allowed for each encountered active object along the active delegation chain, an extra passive object would need to be stored in the active closure, namely the passive (lexical) implementor[5]. This would

[5]The dynamic passive receiver, `this()` would not have to be stored, since it can always be

mean that the lexical behaviour has to be copied to ensure the constraint on remote references, if the active closure would need to travel across the network. Such semantics would obviously be very confusing and expensive. By allowing method invocation on active objects only to consider the direct behaviour, the need for storing passive objects is avoided, since all necessary passive context information can be restored from the stored "current active object". `capture()` and `this()` are derived as the active object's behaviour, `super()` is that behaviour's parent.

The active object delegation algorithm is very briefly touched upon in the semantics of dPico found in appendix A.4.3. Having discussed the method lookup, we will now turn our attention to method application.

6.5.2 Applying an Active Closure

The next step in evaluating `ao.m(arg)` is to invoke the active closure `ao.m` on the proper arguments. Actual arguments must always be evaluated in the context of the caller. For active closures, the binding mechanism is entirely similar to the passive case. The most important distinction with passive method invocation is the question of *who* will evaluate the body of the method. In the passive case, there was only one candidate for evaluation, being the active object performing the call to the passive object. In the case of active objects, we can choose between three active objects to perform the evaluation: the active object performing the method invocation, the active receiver of the message or the lexical implementor of the method.

6.5.2.1 Choosing the Method Evaluator

We have chosen to make the lexical implementor of the method the evaluator of the method body, as it is the most logical choice in the context of lexical scoping. That is, the call frame of the method must *always* extend the behaviour part of this lexical implementor. This is needed to ensure that "free variables" in the method will correctly refer to the lexically visible instance variables of the active object. From a more conceptual point of view, it also makes sense to have the implementor of the method also execute it. A final argument in favour of our choice is that our delegation scheme therefore comes very close to Henry Lieberman's original proposal of delegation in (Lieberman, 1986). There, objects or actors can delegate a message to their *proxy* (their parent in our specific case), thereby implicitly stating "I don't know how to respond to this message, can you respond for me?" (Lieberman, 1987).

This kind of delegation scheme is very close to ours, except that we have to break down delegation in three steps: method lookup, argument evaluation and method application. The reason why the arguments cannot just simply be evaluated first and then passed on to parent active objects together with the delegated message is that some care has to be taken with Pic%'s call-by-name parameter

derived as the behaviour of the current active object.

passing semantics explained in section 2.5.3.2. By introducing lazy evaluation, it must first be ensured which arguments may be evaluated before telling a parental object to apply a certain method. Nevertheless, the idea of having a parental object handling the message request instead of the original receiver remains very close to the original idea of delegation. Late binding of self is translated to late binding of active self in the context of active object delegation. That is, during method application, `athis()` will still point to the original receiver, not to the lexical implementor. `this()` however, will always point to the behaviour of the lexical active object. We cannot make `this()` point to the behaviour of `athis()` as this behaviour could possibly be located remotely, and we cannot create remote references to passive objects.

6.5.2.2 Method Evaluation Context

When the arguments are finally evaluated and bound to a call frame, all that remains to be done is to pass on this call frame to the lexical active object `ao'`. This is done by scheduling the call frame, the method body and a promise in the active object's request queue. The caller then merely continues its evaluation, evaluating `ao.m(arg)` to a promise. When `ao'` processes the request, it will set the call frame's pointer to its behaviour and will start evaluating the method body in a context where `this()` is bound to his behaviour, `super()` is bound to the behaviour's parent, `athis()` is bound to `ao` and `asuper()` is bound to `ao'` its parent. This evaluation context is visualized in figure 6.11.

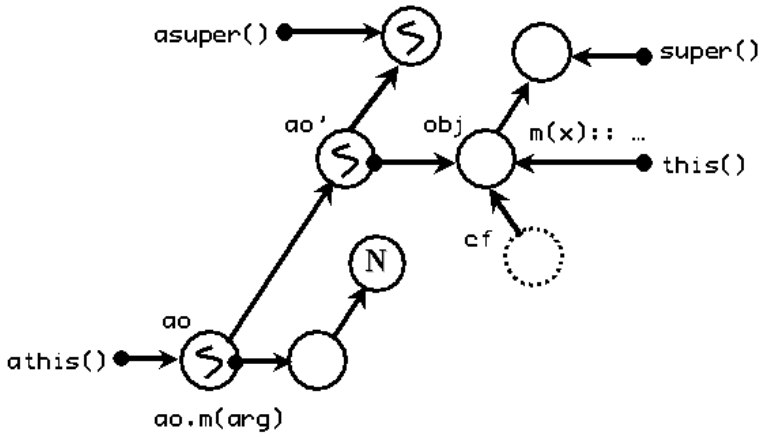

Figure 6.11: Active Object Method Evaluation Context

When the method's body is evaluated, the corresponding value is used to fulfill the promise that was passed by the caller. This will wake up any active object blocking on the promise. More detailed semantics regarding active object lookup, active closures and active object method invocation can be found in appendix A.4.3.

6.5.3 Minimizing Network Traffic

Performing method lookup over the network is a costly operation, which is why such lookup should only be performed when it is essential for the correct operation of the program. In order to minimize unnecessary network traffic, the active object hierarchy can be "collapsed", thereby obtaining locality. dPico supports such a collapse through the `copydown` native.

Figure 6.12 shows how a copy of the behaviour that is being "copied down" replaces the natives dictionary[6]. The natives are typically the parent of the behaviour of an active object, although this is not always so due to imperative mixin methods (see section 6.1.3.2).

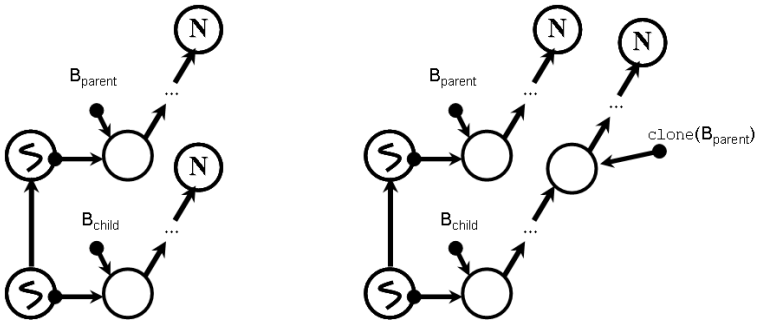

Figure 6.12: Applying `copydown`

Let us introduce the use of `copydown` using a simple program that allows travel agencies to book airline company flights. The code fragment below shows a constructor function for an active view per airline company. Every airline company exports two simple methods, which are not further specified here. The `book` method books a flight to a given destination for a client, and the `list` method returns a table of all destinations that this particular airline offers. The `travelAgency` constructor function represents a travel agency's view on the particular airline company.

```
‘ destinations is a table of passive objects ‘
‘ with a ’name’ and a ’book’ method ‘
aview.airline(destinations) :: {
  book(name, client) :: ...
  list() :: ...
  aview.travelAgency() :: {
    copydown();
    list() :: .list();
    update() ::
      destinations := asuper(destinations)
```

[6]Since a copy-down involves the entire behaviour of the parent, the natives will still be at top of the hierarchy afterwards.

```
  }
}
```

Intuitively, the most frequent request that will need to be answered is a `list` request, since most customers will first want to browse all the possible destinations before deciding to book a flight. The goal is now to reduce unnecessary network traffic to the airline company object. Therefore `copydown()` is used to copy the entire behaviour of the airline to the machine of the travel agency. Thus, the behaviour of the `travelAgency` active object now has a parent which contains `destinations`, `book`, `list` and `travelAgency`. Since the whole behaviour has been copied we can rest assured that all needed local variables in methods are available.

In section 6.5.1, we have already stated that during active method lookup, only the behaviour of the active objects along the active hierarchy are searched. Since the copied-down methods are not expanded in that behaviour, but rather at least one level higher up, an invocation of `aTravelAgency.list()` will actually still travel up to the `airline` object because it is not directly found in `aTravelAgency`. To prevent this, definitions that need to remain visible in the interface of the active object have to be explicitly overridden. This is illustrated in the example where `list` is overridden using `list():: .list()`. This makes `aTravelAgency.list()` find the "artificially overridden" `list`. When evaluating `.list()`, the *local* hierarchy of the travel agency will be searched and the copied down local `list` will be found and applied.

Notice that clients invoking `list` on a travel agency will receive information that is constructed from possibly outdated local information. In the example, this is no real issue since the destinations that an airline offers do not change frequently. Since `book` is not "artificially overridden", `book` messages are always sent directly to the airline to avoid e.g. double bookings. Thus, the method of "artificially overriding" methods can also be seen as an advantage, since it offers some control as to which methods may be used by directly clients. Finally, the example also shows how to update copied-down information in a very simple way using the active scope functions discussed in section 6.4.2.

6.6 Service Discovery: First Contact

In a distributed setting it is essential to be able to get an initial remote reference to an object. As mentioned in section 6.2, such initial references are necessarily always references to active objects. To allow active objects to make themselves visible we have introduced the `ao.register(channelName)` native. This native function ensures that its receiver is an active object, and subsequently registers the active object on a specific channel. The modelling of `register` as a method instead of an operator `register(ao,channelName)` ensures Extreme Encapsulation of objects. Objects not willing to be publicly visible can override `register`.

In our current distribution model, a channel is nothing more than a text string that can be used to describe some properties of the registered object. For example, an active object that represents a data projector might be published on the "projectors" channel. Since it is possible to register an active object on multiple channels, a single object can perform multiple roles. For example a laptop can register both as a "Computer" and as a "Visualization Device".

Using the `register` native we can couple an active object to a channel. In order to retrieve, possibly remote, references to the objects of a channel we use a simple `members` native. This native function takes a channel name and returns a Pic% table of the active objects that have registered themselves to that specific channel. The actual implementation of this native relies heavily on how the channel information is maintained.

Initially the mapping of channels to their members was maintained on a specific server, and all registered object references were thus transmitted to that server. This model was very simple and allowed for the members of a channel to be simply fetched from this central server. However, using a single server is not really wanted in the setting of ambient intelligence. For example, Maria will not want to connect to some server to find out that there is a projector in the seminar room. Rather, devices seemingly "smell" one another as soon as they get into one another's range. The use of a single server also poses severe fault-tolerance problems. If the server crashes, channels would become unusable.

Therefore our current Java implementation, briefly described in section 6.12, does not use a global server but rather keeps published remote object references local to each virtual machine. Every VM contains a mapping per channel for all active objects that have locally registered to that channel. This has the advantage that such a network lookup is a lot less vulnerable to failures, since the management is no longer in the hands of one server. Moreover this approach is also better suited to model devices that may go out of range in a wireless setup. At this point they will appear to be no longer present on the channel.

To find the members of a specific channel, a single request to some server is now impossible. Instead, a request for the members of a given channel is broadcasted to all devices currently "in range". Each such device responds to the broadcast message with the set of active objects locally registered in that channel. After this handshake with all available members, the sets are unified and the result converted to a fixed-size table. This mechanism is illustrated in figure 6.13. The implementation makes some simplifying assumptions, such as the fact that all virtual machines in range will properly reply to the request.

In the future we would like to explore what role multivalues, which are a lot more dynamic than Pic% tables, can play in this discovery process. We could for example return a "multivalue" before all nodes have returned, where results are "accumulated" as they arrive at the node. This would also avoid the requirement that all process should return a value. However we have explicitly chosen not to incorporate multivalues as of yet in dPico to be able to focus on the core of the language's concepts, such as distributed hierarchies. We have deferred multivalues

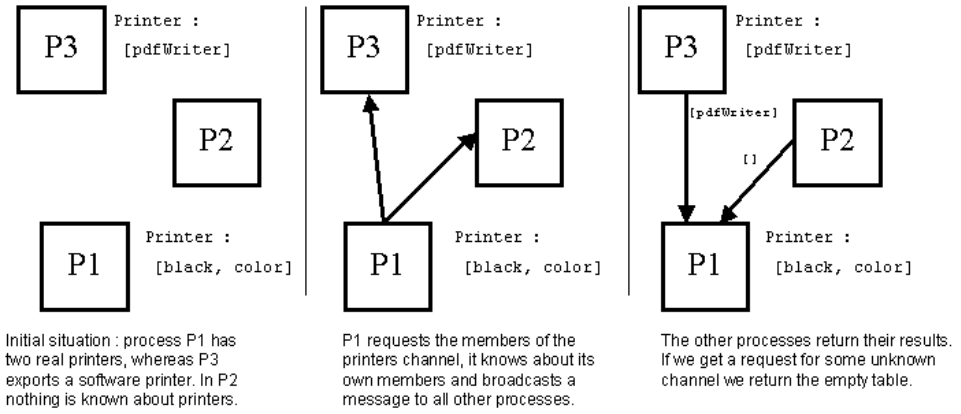

Figure 6.13: Broadcasting a `members` request

as future work.

6.7 An Example: a Distributed Chat Client

Up to now, we have defined several concepts in dPico, including a new object creation mechanism, a distinction between an active and a passive object hierarchy and a plethora of `this` and `super` constructs to access both hierarchies. This section is meant to bring it all together and to give an example of how distributed applications can be written in dPico.

The example illustrated here is a distributed chat client. It uses parent sharing as a primary mechanism to structure the code. It also shows how parent sharing is used achieve collaboration between several child objects. Due to the distributed nature of the application, all objects in the example are active. A *server* object is introduced which is designated two tasks. First, the server decides who can join a given chat channel. Second, it is the server's responsibility to *broadcast* a message to all registered clients. The server is modelled as a *shared parent* of the registered clients. The clients themselves are active views on the server. A general structure of the code example is shown below to emphasize the parent-child relation of server and clients.

```
aview.chatServer(channelName, maxClients) :: {
   clients[maxClients] : void;
   occupancy : 0;
   aview.registerClient(nam) :: {
        ...
   };
   sendMsg(msg) :: { ... };
   ...
};
```

The server is defined as being an "active view" on the top-level active object (*main*). The `registerClient` method can be used to create active views on the server. It will also *automatically* register the client it creates with the parent. If `registerClient` is called remotely, it will naturally create a netview (i.e. a remote active view) on the server. The server contains a table of to keep track of registered clients and a variable keeping track of the *number* of registered clients.

Upon initialization of the client, the parent needs to register the new active view. This is plainly done by adding the view to the `clients` table. However, how is the parent to get a reference to its own active view? Upon execution of `registerClient`, the `aview`'s code is already evaluated in the context of the client itself. Therefore, the client will perform the registration inside the body of `registerClient`. Note that it can be *guaranteed* that any created client will be properly registered, because the client code is nested in the parent's code. Hence, the server dictates the behaviour of its clients, and malicious behaviour is ruled out. `registerClient` is defined as follows:

```
aview.registerClient(nam) :: {
   receiveMsg(from,msg) :: display(from,": ",msg,eoln);

   asuper(
     if (occupancy=maxClients,
         error("Sorry, channel is full"),
         clients[occupancy := occupancy+1] := athis()) )
};
```

The method `receiveMsg` will be invoked by the server whenever a client has sent a message to the chat channel. A client will plainly print this message to the screen, together with the identification of its sender. More important is the use of the active scope function `asuper` to register the client with the server. Recall that `asuper` asynchronously evaluates the argument expression in the context of `asuper()`. The latter denotes the active object under extension, which is the server in this example. Hence, the client politely registers itself with the server. The server does not have to define any special method for it. This registration mechanism is *secure*. If the server would have had a public method `register(aClient)`, malicious objects could use such public method to register client objects that are no children of the server.

Notice how late binding of active self is exploited in the registration code. The object that gets added to the `clients` table is `athis()`, which still denotes the *active view*. Hence, the server is properly passed a reference to its client child. What follows is the code for `sendMsg`, defined on the server object, which broadcasts a message to all registered channel clients.

```
sendMsg(msg) :: {
   from: athis(nam);
```

```
    for(i:1, i <= occupancy, i:=i+1,
        clients[i].receiveMsg(from, msg));
    "message sent"
};
```

Although the message is defined in the parent, it is meant to be invoked on a client. Delegation will ensure the message is understood. As explained in section 6.5, a message delegated to a parent will also be *executed* by that parent. Thus, it is the parent which will execute `sendMsg`. Due to late binding of self, `athis()` will refer to the client to which the `sendMsg` request was really sent. The parent can use the active scope function `athis(exp)` to retrieve the *name* of the sender (which is the initial receiver of `sendMsg`). The server then reroutes this message to all of its registered clients by sending each of them the `receiveMsg` message. Notice that such invocations happen *asynchronously*. This illustrates the power of asynchronous message passing, where the calling object (the parent) can quickly distribute messages to possibly very distant chat clients without suffering from delays. `sendMsg` will probably return even before the text message is received by all clients.

It should be noted that the message `sendMsg` should not be sent directly to the server object. Doing so would deadlock the server: `athis()` would point to the server itself, so the server would asynchronously ask itself for its name and block forever on the resulting promise. This is because the request fulfilling the promise returned by `athis(nam)` is scheduled in the server's own queue. When the server subsequently touches the promise, it blocks and thereby keeps the scheduled request that should fulfill the promise from getting executed.

To be able to access the chat server remotely, service discovery as explained in the previous section should be added. To this end, a chat server will register itself on the dPico channel denoted by its channel name. It suffices to add the expression `register(channelName)` to the body of `chatServer`. Since a function application of `register` is considered similar to a self send `athis().register`, it is the chat server under creation that will be registered.

Finally, consider the following code being executed on a virtual machine *different* from the one on which a chat server hosting the channel "vubServer" is deployed.

```
{ vubserver: members("vubServer")[1];
  client : vubserver.registerClient("John Doe");
  client.sendMsg("Hello World") }
```

A remote reference to the server is retrieved by requesting the members of the "vubServer" channel. `client` is a remote active view on this server. The message sent to this client will properly be broadcast by the parent to all registered clients on any number of virtual machines.

The distributed chat client example illustrates how easy it can be to structure software according to a simple parent sharing relation. A non-trivial client-server

application as outlined above can be written in dPico in less than 20 lines of code which we think is quite expressive. Bidirectional communication between parent and children is naturally achieved through the delegation link. Upward communication happens through implicit delegation or explicit super sends. Downward communication is made possible by late binding of self. A similar example making use of such communication can be found in (Dedecker and De Meuter, 2003).

Notice that all participating objects in the example are active. This means that they find themselves protected against race conditions, since messages sent to them are properly serialized. This eases reasoning about the concurrency involved. The programmer will still have to be careful with the return values of asynchronous message sends, however. The complete source code of the `chatServer` example can be found in appendix B.2.

Having discussed the most important concepts of dPico, the following section will discuss promises, inherited from cPico. An implementation-level technique will be discussed that keeps promises a manageable abstraction in a distributed context.

6.8 Promises in a Distributed Context

In section 5.3.3.2, we have introduced promises as a mechanism for supporting return values of asynchronous method invocations on active objects. In this section, we will extend the use of promises to support method invocations on remote active objects. Although the concepts will stay largely the same for the dPico programmer, we will need some extra support to ensure a correct operation of promises in a distributed setting. To sketch the problems involved, notice that a message sent to a remote object will be handled by the receiver itself. This means that the method corresponding to the message is executed remotely (from the point of view of the caller). A promise is used by the caller to get a handle to the result of the invocation. However, the callee will also need a reference to the promise to be able to fulfill it. Thus, we have two active objects on separate virtual machines each needing a reference to the same promise.

This introduces conceptual as well as implementation-level problems. Conceptually, the only language value that could be shared between virtual machines up until now were active objects. Promises will have to break this rule such that callees can properly fulfill the correct promise. The following section will explain how promises are adapted to become remotely accessible while hiding this from the programmer. Section 6.8.2 will then discuss *automatic continuations* (Ehmety et al., 1998). These continuations allow for increased asynchronicity using promises and are related to the implementation-level problems promises raise in a distributed context.

6.8.1 Remote Promises

As mentioned in the introduction, the problem in using promises for remote method invocations is that the promise itself needs to be *shared* between multiple interpreters. In order to deal with this sharing, we introduce a *remote promise*, which is nothing more than a proxy for a promise residing on a different machine. When transferring promises across a network, we will not send the promise in itself, but rather a surrogate, a remote promise. Such references are very similar to the remote object references to active objects. The idea is that the caller of a remote method invocation gets a reference to the locally created promise, while the callee receives a remote promise. When the remote promise gets fulfilled by the callee, it will automatically *forward* its fulfilled value over the network as to fulfill the local promise. If the calling process was blocked on the local promise, the remote active object can implicitly wake it up through the remote promise. Just as the implementation-level remote object references require a "remote object table" to map proxies to objects, so do remote promises require some bookkeeping by a "remote promise table".

6.8.2 Automatic Continuations

Automatic continuations, as explained in (Ehmety et al., 1998) occur when an asynchronous method fulfills its promise with a *partial* result. A partial result is either an unfulfilled promise or a data structure indirectly pointing to an unfulfilled promise. When the callee returns such partial results to its caller, two strategies can be followed. Either the callee must *wait* for the result to become *complete* (i.e. all unfulfilled promises in the return value must be fulfilled), or it can simply return the partial result already and continue serving other requests. When the latter strategy is chosen, an *automatic continuation* is said to occur.

Such situations can occur in cPico or dPico when an active object `a` sends a message `b.m()` to another active object `b`. `a` will immediately receive a promise that will be fulfilled with the return value of `m`. If `b` now returns a promise as `m`'s return value, `a`'s promise will be fulfilled by `b`'s returned promise. As explained in chapter 5, fulfilling promises with other promises is possible in our language and so `b` can continue processing other requests – it does not have to wait for its return value to be complete in order to "return from the method call". Thus, our semantics and implementation allow for such automatic continuations.

In (Ehmety et al., 1998), both approaches to partial return values – blocking the callee or allowing for continuation – are compared. The automatic continuation strategy is preferred since it exploits more parallelism by favouring chained asynchronous method invocations. If an active object `ao1` invokes a method `ao2.m()`, and this method returns `ao3.n()` as its value, then it makes sense that `ao2` can continue and that it does not have to wait for `ao3` to compute `n` before it can return its value to `ao1`.

6.8.2.1 Distributed Automatic Continuation Support

A distributed context that necessitates the use of forwarding promise proxies complicates the support for automatic continuations. To see why, reconsider the example of the three active objects `ao1`, `ao2` and `ao3` from the previous section. Assume each of the objects is located on a different virtual machine. These objects and the methods they invoke are shown in figure 6.14. The method invocation `ao2.m()` initiated by `ao1` will spawn a local promise ρ_{l1} at `ao1`'s location and a remote promise ρ_{r1} at `ao2`'s location. Likewise, the call `ao3.n()` generates promises ρ_{l2} and ρ_{r2}. Now, the promise ρ_{l2} will be the return value of `m`, and thus ρ_{r1} will be fulfilled with it. Since ρ_{r1} is a "forwarding proxy", it forwards ρ_{l2} to ρ_{l1}. Recall that promises are never transported across the network. Instead they are passed by reference and thus a new remote promise ρ_{r3} is created at `ao1`'s site, "pointing to" ρ_{l2}. If `ao1` now blocks on ρ_{r3}, it will *again* have to perform a remote call to `ao2`'s interpreter to see whether ρ_{l2} hasn't been locally fulfilled yet. The example is visualized in figure 6.14. Full circles represent local promises, while dotted circles represent remote ones.

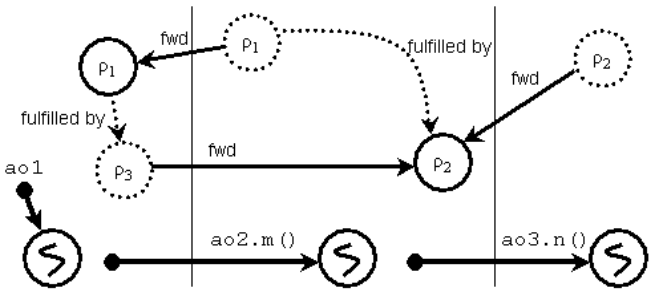

Figure 6.14: Using Remote Promises to support Automatic Continuations

To eliminate this excessive remote promise and the resulting network traffic overhead, we employ an algorithm that will transform "fulfilled by" links into "forwards to" links. Whenever a promise π is fulfilled by another promise ρ, we will not "propagate" ρ, but instead ask it to *forward its value* to π when it becomes determined. Thus, we have turned ρ into a "forwarding promise" which will forward its eventual value to π. Thus, the relation "π is fulfilled by ρ" is reversed and transformed to "ρ forwards its value to π". Figure 6.15 illustrates the changes this implies for the previous example. Notice the disappearance of ρ_{r3}. This promise is no longer needed as ρ_{r1} will only forward *determined* values over the network. Notice how a "forwarding chain" is created naturally, flowing from one promise to another. When the call to `ao3.n()` finally returns some value, this value will travel all along the forwarding links, fulfilling the promises as it is passed on through.

If a remote promise is now asked to forward another promise, the remote promise asks that promise to forward its eventual value whenever it becomes available. To ensure that this value is in turn not a promise, the forwarding scheme must be

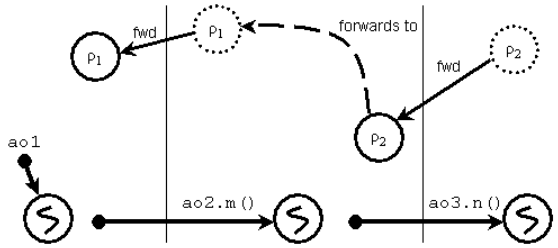

Figure 6.15: Using Promise forwarding to support Automatic Continuations

applied recursively. Since a local promise can now become a "forwarding promise" as well, we must encode a general mechanism for local as well as for remote promises.

6.8.2.2 Promise Forwarding Graph

From the above example, the reader might get the impression that a forwarding promise ρ can only ever forward to *one* other promise π. But, since "ρ forwards to π" is now equivalent to "π is fulfilled by ρ", we see that this is only true if there is *no other* promise π' such that "π' is fulfilled by ρ". This condition does not hold in general: we can have one value that fulfills many promises. A really easy way to accomplish this is to make use of call-with-current-promise as explained in section 5.6.2. Using the `delay` native, it becomes possible to get hold of promises as manipulable values. It is then easy to write:

```
fulfill(p1,p3);
fulfill(p2,p3);
```

Here, `p3` corresponds to ρ, while `p1` and `p2` represent π and π'. Since there are now *two* promises dependent on the value of `p3`, the latter must forward its eventual value to *both* of them. This promise forwarding scheme is easily mapped on a graph $G(V, E)$ where the vertices V are promises and the edges E can be defined as $\{(\pi, \rho) |\ \pi \text{ forwards to } \rho\ \}$. This graph can easily contain cycles, as is witnessed by the legal expression `fulfill(p,p)`. If an active object ever touches a forwarding promise, it will have to block and wait for the promise to be fulfilled, as is the case for regular promises. Thus, although G may contain cycles, touching a promise in a cycle will not lead to a livelock by endlessly chasing forwarding pointers since forwarding promises *only* forward fulfill requests, *not* "touch requests".

When a forwarding promise gets fulfilled by a determined value (i.e. any value *other than a promise*), it will transcend to a determined state and forward its value to all of its "registered clients". From that moment on, the promise is no longer a forwarding promise. If a number of forwarding promises are structured in a cycle, and one of them gets fulfilled with a determined value, interesting behaviour

occurs. The fulfilled promise will forwards its value to its neighbour, which will again forward it to his neighbour, and so on.... As such, the entire cycle will all of a sudden get fulfilled. This fulfillment through forwarding will not go into livelock (i.e. the value is not forwarded around the cycle indefinitely), since a promise *first* becomes a determined promise before it starts forwarding. Since determined promises plainly ignore any forwarded *fulfill* request, the travelling value will be "absorbed" by the initiator when it has gone round the entire cycle once.

To summarize, we see that a promise in our improved forwarding scheme can be in one of three states: *undetermined* when it does not have any value yet, *determined* when it is fulfilled by a determined value and *forwarding* when it is fulfilled by a promise itself. Moreover, a promise can be subject to three fundamental operations: it can be fulfilled, it can be asked to forward its value to another promise and it can be queried for its value. The entire state diagram for promises is shown in figure 6.16. In the diagram, we assume p to denote a promise and v to denote any value but a promise. Each transition is annotated with pseudo code indicating the response to a given message in the receiving state.

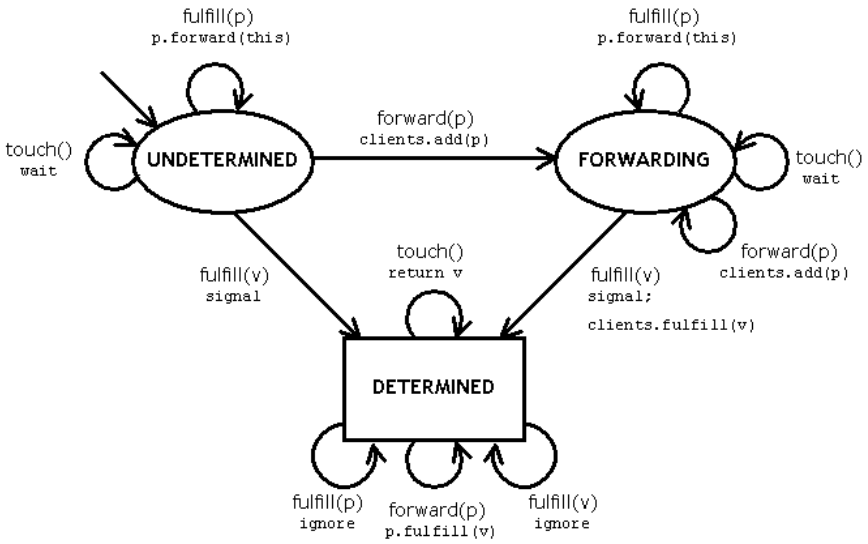

Figure 6.16: The Promise State Diagram

A local promise always starts out in the *uninitialized* state. One can regard a "remote promise" as being nothing more than a local promise that is initiated in the *forwarding* state, having as its sole forwarding client some promise across the network. Remote promises can therefore also be plainly reified using call-with-current-promise.

The scheme of forwarding promises is more than a plain optimization technique to support proper automatic continuations. From the above explanation, it is clear that forwarding promises *avoid livelock*. This would not be the case if we would not employ the forwarding scheme. Consider the execution of the following

code fragment in such a case:

```
fulfill(p1,p2);
fulfill(p2,p1);
```

This would transform both `p1` and `p2` into *determined* promises, rather than *forwarding* promises. If an active object were to perform an operation on one of the promises, it would inevitably get stuck in an infinite loop: `p1` will consider itself fulfilled and will delegate the operation to `p2`, who will in turn delegate the operation to `p1`, ad infinitum. We see that, by turning promises into forwarding promises that do not delegate "touch" requests, we can banish such promise livelocks.

6.9 Transmitting Environments: Basic Mobility

Programs can be transformed to first-class values using `call` as *environments*, a property dPico has inherited from Pico. Such environments are Pico's nomenclature for a run-time stack paired with a dictionary (representing the root of the "heap"). Such values are called *continuations* in Scheme. Because an environment is first-class, a "snapshot of a running program" can be perfectly passed as an argument to a remote method invocation. Recall from section 6.2.2 that any value other than active objects and promises are passed by copy. This means that the environment will have to be copied, which includes the run-time continuation stack of the program and all values reachable from the environment root.

This ability of passing around Pico environments allows for a primitive form of code mobility. It is possible for some active object running at a virtual machine A to execute:

```
... ‘part 1‘
call({ result := remote.interpret(cont);
       continue(exit, result) });
... ‘part 2‘
```

We assume `remote` to refer to a remote active object and `exit` to be bound to some environment denoting a place to "jump to" as to escape the local execution of the code labelled *part 2*. The purpose of the code excerpt is to make a snapshot of *part 2* by storing it in the variable `cont` and to subsequently pass that environment to some object residing on virtual machine B. If the latter object issues a `continue` on the given environment, it will *locally* execute *part 2*:

```
aview.evaluator() :: {
   interpret(program) :: continue(program, void)
}
```

Moreover, useful data can be passed to *part 2* if this code uses the value of the `call` expression, and if `evaluator` continues the program with a useful value. dPico will ensure that the entire stack is properly copied, together with the heap data. Recall from section 5.3.2.2 that environments also store information on locks and their reentrancy status. When transmitting locks across the network, this information is also properly serialized, such that remote continuation of an environment ensures any copied serialized object will be properly locked again. Notice also that any context information of the program, such as where `athis()` and `this()` point to, is transmitted as well. Thus, even if environments are passed around to other active objects, only the active object evaluating the code will be modified. Context parameters will remain unchanged. For instance, in the example, `asuper` will *not* all of a sudden point to the parent of `remote` upon continuation at B.

The purpose of this section was to show that the fundamental basis for strong code mobility is present in our language. Running programs can be "interrupted" (the run-time stack can be frozen in a Pico environment) and the resulting snapshot can be transmitted to a remote object. The latter can continue execution of the program locally. To properly support strong mobility as we have envisioned it in chapter 1, dPico will have to be extended with more high level constructs. Strong mobility could for example be *unified* with the movement of active objects. If an active object would be able to move to another interpreter while running code, code and object mobility would be unified and the number of concepts in the language would be kept minimal. We will not consider such advanced mobility schemes any further in this dissertation.

6.10 Security and Safety Issues

As we have explained before, security and safety are major issues when dealing with distributed computing. Though we have not devoted too many attention to security and safety issues in dPico, we have ensured that at least the objects themselves can take care of security if they need it. This is possible due to certain language features that inherently enhance the security and safety of the system. Examples are *parent sharing* in combination with a *controlled extension mechanism* and the avoidance of operators that break Extreme Encapsulation. The observation that encapsulated inheritance by views and mixins allows for more security and safety was already made in (De Meuter et al., 1996).

Security Although we provide only a public and a private interface to objects, scope functions introduce a form of "protected" interfacing, by allowing children to access functionality of a parent not reachable through plain super sends. The advantage of this protected interface is that it can only be used by descendants. The parent itself does not have to make provisions and does not have to identify which methods may be accessed by a child, as is the case in Java. This allows the parent to make certain methods which manipulate sensitive data private, such that

other malicious objects cannot abuse these methods. The parent always knows that its own children will be able to access it using a scope function. Since the parent decides on which objects can be its children, security is maintainable.

The aspect of security also comes into play when trying to find a good mix between allowing unforeseen extension of objects and security. Consider a passive object that allows extension from the outside (i.e. it allows for the creation of children with arbitrary code). This means that unknown children can override any set of methods and variables. As in cPico, the parent object can stay in control by invoking lexical functions instead of the overridden ones using `m()` instead of an explicit self-send `this().m()`.

dPico introduces a similar technique for the active hierarchy. The technique provides some additional benefits in a distributed setting. First of all, a call of the form `m()` guarantees that the method in the local behaviour will be called, whereas `athis().m()` may invoke a (possibly unsafe) overridden method. More importantly in a distributed setting evaluating `athis().m()` may lead to remote method lookup. If the method is applied as `m()` instead, it is guaranteed that no network traffic will occur (given that `m` is bound to a function). Locality may be considered for optimization concerns, but also helps to reduce an active object's vulnerability to partial failures. Using the `copydown` native which we have explained in section 6.5.3, we have given the dPico programmer some tools to ensure that even functions defined in the parent can be applied locally.

Safety Where security is meant to fend off possible intruders, safety is more of a software engineering concern. As we have already stated in section 4.2.2, safe languages try to "restrict" the impact of an error or a programming bug. This means that we want to minimize the amount of code the programmer should inspect in order to find a bug. Program structure is an important aspect that needs to be taken into consideration in order to be able to confine errors. We have mapped this need for structure onto the lexical structure by reintroducing static scope and by only allowing object extensions through controlled application of special extension functions which automatically set up proper evaluation contexts behind the scenes.

The introduction of the new object extensions has two additional advantages with respect to security. First of all, we cannot suddenly objectify an arbitrarily deep nesting of call frames. With a call to `capture` this could be done in cPico, thereby changing the semantics of for example delegation of variables. Such complex semantics are often obfuscating to the programmer. It may be hard to trace where these implicit transformations affect the programmer's code, and as such these transformations make the language less safe.

The explicit use of pointers compromises safety. Therefore pointers do not exist in the standard Pic% model and dPico does not introduce them either. All types of local and remote pointers are managed by the interpreter. Parent pointers are made immutable and fixed through encapsulated inheritance with views and mixins. An object extension's parent is known at creation time and cannot be changed

later on. Although we lose some of the flexibility of Self and dSelf, immutable parent pointers are vital for both the safety and the security of objects. If a child would be able to change its parent, it could designate any object to become its ancestor, allowing it to abuse scope functions to gain access to the new parent's variables.

The only other type of remote references left in the system are remote promises which can be reified using `delay`. Remote promises are kept identical to local promises for dPico programmers. Thus, a remote promise can never be forced to forward its value to another promise. The fact that all pointers are managed by the system also allows the implementation of (distributed) garbage collection, which also improves safety, as explained in section 4.2.2.

6.11 Limitations

Although dPico offers an original and innovative way to deal with some non-trivial issues like distributed object inheritance in a controlled and expressive way, we have found that the language has some flaws as well. This section highlights the limitations we have identified. Some of these will probably be dealt with as the language is subject to more design iterations. Others will prove to be more fundamental to the language's core concepts. To avoid these limitations, concepts will have to be entirely removed or reworked. Finally, the language lacks some important features we have not been able to design due to time constraints or because they are too complex to deal with all at once. A list of identified limitations follows.

- As we have mentioned in 6.5.1, method lookup in active objects will *only* consider the direct behaviour of active objects. The behaviour's parent is never searched. This is because active object delegation can cross network boundaries, and the passive objects should not be transmitted by copy during method lookup. One way around this is to reintroduce remote references to passive objects into the language, but to hide these references in encapsulated context objects, such that they will never be directly usable by the programmer. Yet, allowing remote references to passive objects breaks down the simple semantics and complicates the implementation.

- The above limitation has some direct consequences for passive mixins. As discussed in section 6.1.3.2, a mixin method will move the passive hierarchy "upward", in the sense that the subject of a mixin will receive a new parent object. Now consider a mixin method applied to the behaviour of an active object. As discussed above, when performing method lookup in active object, only the behaviour itself is visible, not its parent. Thus, only mixed in behaviour will be visible: any method `m` defined in the old behaviour that is not overridden in the mixin suddenly becomes invisible. The dPico programmer explicitly has to override each method he wants to "inherit" from the old behaviour by writing code such as `m() :: .m()`.

- Although the active scope functions `athis` and `asuper` allow for an expressive communication link between child and parent, section 6.4.2 has warned about possible deadlocks that can occur if the promise returned by the scope functions is used unwisely. What is most problematic is that it is not always clear when it is safe to use the return value of an active self-send using `athis`. If `athis()` is not bound to a different object, the sender will deadlock because it will eternally wait for a promise it has to fulfill itself. This problem is, however, not inherent to the scope functions themselves. Rather, it is an instance of the more general problem of a circular wait, incarnated in dPico by blocking on promises. When using such tools to control inheritance, deadlocks can always be willingly or unwillingly programmed.

- In section 6.2.2, it was explained how passive objects are passed by copy during a remote method invocation. Performing such a transitive copy from a given root object is problematic in the context of serialized objects[7]. To properly marshal a serialized object, its mutex must be acquired to ensure that no active object is using the object when it is copied. This may lead to very subtle deadlocks as it is always dangerous to invisibly acquire locks. If such a deadlock occurs, the programmer will hardly be able to debug it, since it requires him to trace the entire transitive closures for cyclic waits.

- The different semantics we have developed for parameter passing to local or remote active objects come at the cost of sacrificing some location transparency, as was already noted in section 6.2.2. Consider an active `library` object and a passive `book` object. The semantics of invoking `library.archive(book)` will highly depend on `library`'s location. If `library` is a local active object, `book` will be passed by reference. If `library` is located remotely, `book` will be copied, and side effects of `archive` will not be visible to the original `book`. The only two solutions are either to pass `book` by copy even if `library` is a local object or to pass `book` by reference if `library` is remote. The first solution is expensive, whereas the second introduces references to passive objects, complicating the semantics.

- Similar problems occur with call-by-name parameter passing in a distributed context. Imagine the invocation of a method taking call-by-name parameters. Since actual arguments always need to be wrapped in closures as to not lose context information, the closure will be passed by copy if the receiver is a remote object. This leads to the copying of the caller's context. Although it is expensive, it is necessary to ensure no passive objects are shared between interpreters. This again prevents remote references to passive objects, ensuring that active objects remain the unit of sharing.

[7] Here, we interpret serialized in its concurrent setting. We will employ the term marshaling for serialization in a distributed context to avoid confusion.

- dPico completely lacks error handling constructs, crucial to any realistic distributed programming language. To construct realistic ambient programs, it would need to deal with such problems as unavailable remote objects and promise timeouts. Incorporating full-fledged partial failure handling mechanisms in dPico falls outside of the scope of this dissertation. We will explore some of our ideas regarding this topic in our directions for future research in chapter 7.

6.12 Proof of Concept Implementation

In order to conduct our experiments and to validate our design decisions we have implemented dPico in Java. Implementing and using several versions of dPico allowed us to test the consequences of alternative language design solutions for important choices, both on the implementation of the semantics itself, and on the usability of the model for the programmer. A clear example of such a decision is the rejection of incremental locking schemes (see section 5.7.1) which lead to the introduction of several new concepts as is described in section 5.2.

The choice for Java as our implementation medium is partly motivated by platform independence and because the Java virtual machine is available on a number of small mobile devices such as cellular phones through the Java 2 Micro Edition application environment (Sun Microsystems, 2004). Our current implementation has not yet been ported to J2ME, but with future development for such target devices in mind we have chosen the Java platform to avoid having to rewrite our existing code-base completely.

Currently our proof of concept implementation of dPico does not use a real network for communication. Instead, a dPico interpreter can start up multiple virtual machines – each having its own front-end window for user interaction. These "distributed" virtual machines have only *one* object in common. This singleton object models a network, through which virtual machines can communicate. Nevertheless we have been careful to never rely on the locality of objects. We use the built-in Java serialization mechanism to encode a dPico object graph in a bytestream. Only bytestreams and similar native Java types can be communicated through the network object. Remote object references are not represented as "proxies" or "wrappers" having a hidden pointer to their referenced objects, but are actually represented as a unique identifier. When passing a remote object reference, only this object identifier is transmitted. A remote object table is used to resolve such identifiers to local objects. This also guarantees a total decoupling of local virtual machines.

The decoupling ensures that no dPico value can ever be really shared between local virtual machines. This should allow us to extend our implementation to a realistic distributed context using only moderate modifications. In an earlier version of the language, we have implemented the distributed virtual machines using SOAP middleware technology. This implementation was abandoned due to prob-

lems in using SOAP in our distributed experiment. The details thereof have been published in (Van Cutsem et al., 2004).

In our implementation we have neither been concerned with efficiency nor with optimization techniques, as language features tended to change during the incremental and iterative development of the language. Moreover, the implementation is intended to serve as a "proof-of-concept" that enables us to test and verify our design, rather than being a full-fledged programming environment. Nevertheless we are aware that efficiency is an important issue, especially when designing a language for small devices with low computing power and stringent memory constraints. Therefore we have never considered language features that would be *inherently* inefficient.

6.13 Conclusions

This chapter has introduced a distributed extension for Pic%, called dPico. dPico inherits concepts from cPico, such as active objects with atomic method invocation, but puts them in a different – distributed – perspective. The key element in this new perspective is the introduction of a new active delegation hierarchy that parallels the passive one. The idea is to completely separate active and passive delegation hierarchies. This separation, along with the fact that only active objects can be remotely referenced, promotes active objects to the unit of distribution.

dPico has also removed cPico's liberal object extension mechanism based on calls to `capture` and `activate`. Instead, a more strict, but also more readable and more structured scheme is introduced. This scheme is based on "lifting functions" such as `view` and `aview`. These particular natives are responsible for the extension of the passive respectively active object hierarchy. In combination with a simple discovery mechanism, this allows us to write programs with distributed object inheritance. This type of applications can already be used to express a variety of distributed applications categorized as "connected applets" (De Meuter, 2004), which is currently often mimicked using a combination of HTML with embedded JavaScript. Regarding strong mobility, dPico allows for programs to travel across the network in the disguise of encapsulated environments. Although the key ingredients to support strong mobility are present, the research to put them all together is deferred as future work.

In order to maintain the benefits of promises for active object method invocation across the network, remote promises were introduced as "network proxies for promises". By exploiting the relation between promises, a more efficient "promise forwarding graph" was constructed, through which only determined values (i.e. non-promises) could flow, avoiding the transmission of promises across virtual machine boundaries.

dPico has been found to have some advantages with respect to security and safety due to its ability to enforce a disciplined use of parent sharing for both concurrency and distribution control. Nevertheless, dPico still has its limitations such

as the limited scope of active object lookup. This lookup scheme only searches the direct behaviour of active objects to avoid having to pass remote references to passive context objects. Furthermore, distribution is not entirely transparent since sharing of passive objects is allowed between local active objects but disallowed between distributed active objects.

Summarizing dPico, we can state it to be a small distributed language, featuring distributed active object inheritance to support sharing of state across virtual machine boundaries. Paired with encapsulated inheritance, this allows for safe distributed applications to be written in a high-level language. Apart from parameter passing semantics, the location of the active objects is made transparent, such that the programmer never explicitly has to deal with remote references, explicit proxies or the concept of a location in its own right. We believe such "location abstraction" to be as important to high-level distributed languages as is "variable address abstraction" to high-level sequential languages.

Chapter 7

Conclusions

As explained in the introduction, the goal of our work was to explore how to write programs that work in – and interact with – a dynamic, flexible, mobile and open computing world. This has lead us to the exploration of a variety of programming languages that are used to write concurrent and distributed programs. However, we feel that none of these languages is suited to writing programs that target the digital habitat of a "processor cloud" in which personalized small devices interact with computers that have become invisible and embedded in the environment.

Technically, this dissertation has addressed a gap in the spectrum of distributed programming languages. We have advocated that prototype-based languages are more suitable for writing distributed applications than class-based languages. Class-based languages suffer from added complexity in a distributed context. One example concerns the movement of objects across the network, requiring a whole hierarchy of classes to travel along. Moreover, these classes should be kept consistent, if language semantics are to remain clear. This places a heavy burden on the run-time system supporting such a distributed language. Following the vision that future computing environments will consist of small devices with limited memory and computing power, it is our conviction that another solution should be explored.

Prototype-based languages, featuring only objects, may provide a solution. The observation that objects without classes can perform better in a distributed context is certainly not new. This approach has lead to the creation of some languages that already implement part of our ideas. These languages, which we have described in section 4, all seem to lack an essential feature. Emerald (Hutchinson et al., 1991) and Obliq (Cardelli, 1994) are both promising languages, but neither of them introduces the distributed delegation we wish to investigate. dSelf (Tolksdorf and Knubben, 2001) does introduce distributed parent sharing, but its support for concurrency is limited and the language seems to have very little concern for security at all.

7.1 Reflections on cPico and dPico

Our approach, embodied in two tiny languages, tries to exploit the concept of shared parents between distributed objects to deal with issues in both concurrent and distributed programming. dPico has been iteratively designed, starting from the prototype-based language Pic% (De Meuter et al., 2003b). Distributed object inheritance was introduced without sacrificing the safety that was offered in the base language through nested mixin-based inheritance (Steyaert et al., 1993) and by following the principles of Extreme Encapsulation (De Meuter, 2004).

cPico unifies the notions of an object with that of an autonomous process into an *active object*. This results in an atomic and asynchronous invocation of methods on such objects. Problems regarding the lack of synchronization or return value with asynchronous invocations are countered by the use of *transparent promises*. These promises allow for asynchronous invocations to have a return value just like synchronous ones. The main idea of the concurrency model is centred around the question whether parent sharing facilitates consistent shared data between autonomous processes. cPico demonstrates how awkward it is to share passive data through composition relations and how parent sharing may be a viable alternative. Promises are used as a means to introduce conditional synchronization, keeping the language minimal since it does not require the addition of new concepts.

The concepts of cPico have been extended in a distributed setting, leading to the language dPico. The active objects introduced in the concurrency model have become the unit of distribution. The rationale is that distributed objects should beware of concurrency issues and should favour asynchronous communication, two concepts taken care of by active objects. To impose a program structure where a clear-cut distinction between active and passive objects can be made, both types of objects are organized in a separate delegation hierarchy. This leads to symmetrical active and passive object extensions. Such an organization ensures that all message sends to remote objects are asynchronous.

With the introduction of dPico we have created a prototype-based language which uses parent sharing to express concurrency and distribution. To the best of our knowledge this type of language has not been thoroughly explored thus far. Figure 7.1 categorizes some of the languages we have discussed in the dissertation according to the concepts of prototypes, delegation, concurrency and distribution. We consider Java to be a language that addresses both concurrency and distribution, due to its support of remote method invocation. Borg and ABCL/1 are classified as prototype-based languages even though they are actually more object-based since they lack a delegation mechanism.

7.2 Rough Edges to The Proposal

This section describes the key points in our research that have not yet been entirely resolved. It also reviews the deficiencies of the proposed languages and how these

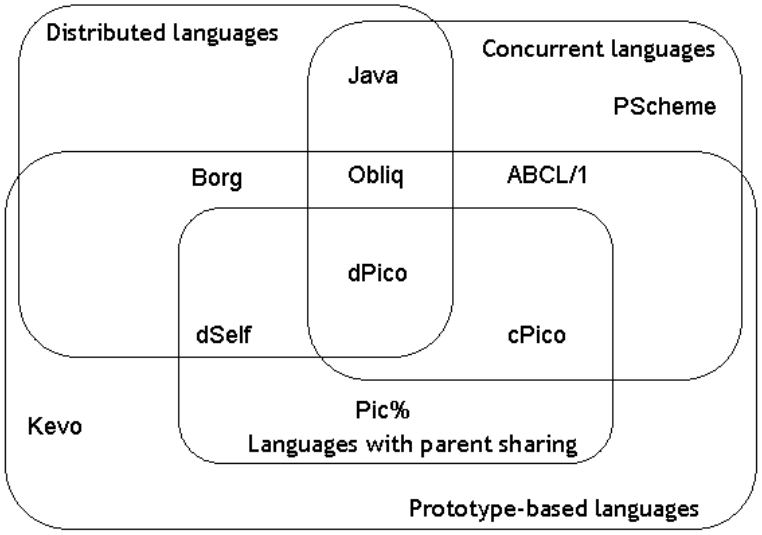

Figure 7.1: Categorizing Languages According To Key Language Features

deficiencies should be removed in future work. The research areas of importance that were left unattended in this dissertation will be briefly recapitulated in the following section.

As mentioned in section 6.11 of the previous chapter, dPico still has its limitations. For example, a method lookup along an active object chain will only consider the *behaviour* of the active objects encountered. Secondly dPico still employs different parameter passing semantics for message sends to local and remote active objects. Both limitations stem from disallowing remote references to passive objects. Perhaps a more nuanced scheme can be constructed that can deal with these issues *without* sacrificing the simple semantics.

Regarding cPico, we we had to thrash out the question of combining delegation with synchronization (see section 5.9). cPico's solution to the problem employs locks: objects can be declared serialized, guaranteeing atomicity through locking. This scheme has the drawback of quickly introducing subtle deadlocks in code where multiple active objects tightly communicate, even when they are structured in a parent-child relation. Perhaps the simple reception queue of active objects is too simplistic and should be augmented with such features as ABCL's selective message reception.

From a technical point of view, one of the rough edges to our work is that it has not been evaluated in a realistic setting. To do so, the proof of concept interpreter will have to be founded with a realistic distribution layer, capable of making two virtual machines communicate across a network. Technologies such as JXTA (section 4.2.4) could be considered for the creation of a more realistic service discovery mechanism. The interpreter should also be ported to small mobile devices, e.g. by making use of the J2ME application environment. Experiments could then

be conducted where multiple interpreters collaborate on multiple mobile devices, interconnected by a wireless network. To be able to conduct experiments on such lightweight devices, the interpreter will have to be considerably optimized. Although we owe much of our research results to Pic%'s simple object model, the representation of objects in dPico is too heavy and method lookup is too slow. It would be interesting to investigate the applicability of Self's object model and adaptive optimization techniques (Smith and Ungar, 1995) to Pic%.

Only very recently, we have come across a distributed programming language very similar in spirit to dPico, called E (Open E Project, 2004). Due to time restrictions, we were unable to thoroughly present the language in chapter 4. The language features an object-centred approach, also using promises and asynchronous message sends. E is worthwhile investigating, as it incorporates a deadlock-free *promise-pipelining* architecture (Stiegler, 2000), which may solve the delegation versus synchronization problem. Other strong points of E are its powerful security mechanism and its fault tolerance abstractions.

Finally, and no less important, we should be gaining some *experience* with dPico by using it to build a variety of distributed applications. Only then will the theory on structuring software through parent sharing be applied in practice and can the language design choices be properly evaluated.

7.3 Directions for Future Research

This section will briefly touch upon some research areas we have not been able to consider in depth but which seem indispensable in a realistic language targeted at Ambient Intelligence. dPico still misses some important language constructs to deal with partial failures and exceptions. Such issues as mobility and broadcasting identified as important language features for AmI in the introduction have also not been dealt with. On a more technical track, distributed garbage collection is left unimplemented in the system. What follows is a brief overview of the most important topics deserving attention in future research.

7.3.1 Split Objects

We have not yet been able to research in depth how dPico's object model could facilitate the management of split objects (Bardou and Dony, 1996). Such split objects occur naturally in frame-based object models where a *single* logical entity is represented by *multiple* physical objects. Such representation is for example used to model *roles* with the typical example of a person that can be regarded *as a* sportsman or *as an* employee. The problem is to find the right abstraction to represent the split object identity. Perhaps active objects that encompass a passive hierarchy could prove to be a solution. The passive hierarchy would be the split object and the encompassing active object would be the identity of the split object.

An issue left unresolved in the same context is the exploitation of the *behaviour*

of an active object. If an active object could change its behaviour, this could lead to interesting abstractions such as conditional synchronization schemes reminiscent of behaviour sets (Kafura, 1990) or a *become* operation similar to the one found in actor languages. One could e.g. model a queue as a split object having the roles or states *empty* and *full*. Messages can then be synchronized according to the "current role" (i.e. the behaviour) of the split object.

7.3.2 Partial Failure Handling

One of *the* major issues left untouched in dPico is partial failure handling (section 4.2.8). To facilitate fault tolerant programs, able to deal with parts of the program crashing or becoming temporarily unavailable without warning (e.g. due to devices going in and out of broadcast range), the necessary language features will need to be developed. There is a strong need for an expressive *exception handling* mechanism. Simple schemes such as the `try-catch` constructs from Java or C++ are not sufficient in dPico, because of *asynchronous* method invocations. More flexible mechanisms are necessary. Emerald's handlers and E's `when-catch` construct (specifically designed for dealing with promises) provide hopeful abstractions.

One simple approach to fault tolerance we have not been able to implement yet is the "caching" or "buffering" of messages sent to temporarily unavailable remote objects. Since our language model ensures that remote objects are always active, these cached messages can simply be "appended" to the active object's message queue whenever it becomes available again. Also, the idea of *sending a message to a promise* was considered, to speed up program execution. Currently, sending a message to a promise will block the sender until the promise is resolved to an object. Some of these ideas appear to have already been developed in the E programming language, which features an *eventual send* operation, able to send messages to promises (Stiegler, 2000). The E language kernel will ensure that messages sent to such promise will be sent to the underlying object, preserving order.

7.3.3 Multivalues

In the course of our scenario we have illustrated the need for broadcast communication. Currently this need is left unaddressed in the proposed model. To support such broadcasting the concept of a *multivalue* – a value which is a number of values all at the same time – is interesting. Locally, a multivalue can be seen as a collection of values, which broadcasts messages to all of its components. Furthermore it should be possible to distribute a multivalue across several virtual machines, where every node maintains its local members. Messages sent to such a multivalue will be applied to the local members and are simultaneously broadcasted over the network where other virtual machines will send the message to their own local members. This mechanism is very similar to the way we currently perform a query with the `members` native (see section 6.6).

Multivalues could even serve as a replacement for our current channel abstraction altogether, where what we now call a channel then becomes a distributed multivalue. Multivalues clearly have potential, but it remains to be seen what the repercussions of their introduction would be on the language.

7.3.4 Mobility Abstractions

One important aspect of Ambient Intelligence that has not yet been dealt with thoroughly in our proposed languages is mobility. Both object mobility as employed in e.g. Emerald and strong code mobility as it has been implemented in e.g. Borg deserve some attention. As explained in section 6.9, dPico is equipped with a good basis to support strong mobility. It remains to integrate it more closely in the language, perhaps by unifying strong mobility with the movement of active objects. Viewed in this light, our active objects are evolving towards full-fledged *software agents*. Indeed, like an agent, an active object is an autonomous computing entity. By properly integrating mobility in the language, such a unification would augment the expressivity of dPico considerably, without having to give up its simplicity.

7.3.5 Distributed Garbage Collection

In order to support the realistic development of truly distributed programs that operate in a real networked context, distributed garbage collection will have to be implemented. More specifically, a garbage collector is needed that takes into account the effects of objects that can move across a network. Objects that may temporarily become unavailable due to nodes going out of range must also be considered. Finally, it remains a challenge to discover how garbage collectors can and should interact with objects that are part of a distributed multivalue.

Appendix A

Pic% Semantics

This appendix goes into more detail on the programming language Pic% and its concurrent and distributed extensions. We describe the semantics of the language using a pseudo-formal model, largely based on our own interpreter, written in Java. Although we do not claim this is a true formal definition of the language, we do think expressing the evaluation rules in this way offers the advantage of readability and clarity. A Java implementation is obviously too complex to discuss in detail. Moreover, the semantics are sketched using *recursive* evaluation rules, whereas the real Pic% interpreter is written in continuation-passing-style, explicitly representing the Pic% run-time stack without allowing (Java-)recursive evaluation. Proper denotational semantics would evaluate expressions with respect to some continuation κ and an environment or store σ. We simplify matters by sticking to an ordinary recursive interpreter, omitting κ, and allowing for our environments to be mutable, thereby making it unnecessary to continually pass updated environments via σ.

The appendix starts with basic evaluation rules for the basic Pic% language in section A.1. In sections A.3 and A.4, this basic language model is adjusted and minimal semantics are provided for some language features of cPico and dPico. Note that it is always our intention to provide *clear* rather than *sound* formal semantics.

A.1 Basic Pic% Semantics

Our semantics start by giving an overview of the *structure* of the Pic% language values. Section A.1.2 continues and defines the rules to evaluate expressions into values.

A.1.1 Abstract Grammar Entities

First and foremost, this section defines what *kind* of values inhabit a Pic% program. That is, we will describe what constitutes a first-class Pic% value. Recall that even

Pic% parse trees are first-class values. These parse trees are the skeleton of a program. Evaluating such structure usually results in a new kind of Pic% value.

Each basic value is represented as a functor, acting as a data structuring mechanism. Each such functor belongs to a certain set or *class* of Pic% values. The fact that these sets are reflected in the semantics is not surprising, as we have implemented the interpreter in a *class-based* language. Hence, any Pic% value was implemented as an instance of a Pic% value class. These classes are structured in an inheritance relation, which we model here using the subset relation. That is, if B is a subclass of A, and $i(C)$ denotes the set of objects which are "instances of class C", then it holds that $i(B) \subseteq i(A)$.

Table A.1 gives an overview of all abstract grammar entities paired with their corresponding set.

Name	**Functor**	**Set**
Number	$nbr(n), n \in \mathbb{N}$	$\mathbf{NBR}$
Fraction	$frc(f), f \in \mathbb{R}$	$\mathbf{FRC}$
Text	$txt(s), s \in String$	$\mathbf{TXT}$
Table	$tab(a), a \in Array$	$\mathbf{TAB}$
Void	$voi()$	$\mathbf{VOI}$
Reference	$ref(txt)$	$\mathbf{REF}$
Tabulation	$tbl(inv, exp)$	$\mathbf{TBL}$
Application	$apl(inv, exp)$	$\mathbf{APL}$
Message	$msg(dct, inv)$	$\mathbf{MSG}$
Super	$sup(inv)$	$\mathbf{SUP}$
Definition	$def(inv, exp)$	$\mathbf{DEF}$
Declaration	$dcl(inv, exp)$	$\mathbf{DCL}$
Assignment	$ass(inv, exp)$	$\mathbf{ASS}$
Quotation	$quo(exp)$	$\mathbf{QUO}$
Function	$fun(nam, arg, bdy)$	$\mathbf{FUN}$
Closure	$clo(fun, ctx)$	$\mathbf{CLO}$
Native	$nat(nbr)$	$\mathbf{NAT}$
Dictionary	$dct(cst, var, nxt)$	$\mathbf{DCT}$
Binding	$bnd(nam, val, nxt)$	$\mathbf{BND}$
Environment	$env(c, ctx), c \in Cont$	$\mathbf{ENV}$
Context	$ctx(cur, ths, sup)$	$\mathbf{CTX}$

Table A.1: The Pic% basic language values

Aside from these values, we also define $\mathbf{AG}$ to be the union of all value sets. We also define invocations $\mathbf{INV} \equiv \mathbf{REF} \cup \mathbf{TBL} \cup \mathbf{APL}$. The set of values $\mathbf{VAL}$ are all values except for the values from the second category. They denote any Pic% value that is *self-evaluating*. The set of qualifications $\mathbf{QUA}$ is equal to $\mathbf{MSG} \cup \mathbf{SUP}$. Not surprisingly, all these special sets are implemented as abstract classes in the interpreter, since they allow for some abstraction over all constituent

subclasses.

There are some remarks to be made regarding some language values. First of all, the class $String$ simply denotes strings of ascii characters (although not all ascii characters are actually allowed). We assume this class has defined an equality operator $=$ over its values, comparing two strings. The $Array$ class simply denotes an indexable data-structure. We assume an accessor and mutator operation, able to get or set the contents of a location in the array. They are accessible through the methods get and set defined on tables. $Cont$ denotes the set of "continuations". A continuation can be regarded as a "frame" containing some data and code. The run-time stack of a Pic% program is entirely comprised of such frames. In our implementation, such frames are first-class objects, although this does not necessarily have to be the case. What is important, however, is that they are made *explicit* in an implementation.

Concerning Void, we will denote it's sole instance by **void**. Natives have a constituent "index", making it possible for the implementation to link a certain native to some implementation-level code. To support the native `call`, we would have to express the semantics using continuation-passing-style, where the "current continuation" is made explicit so that it can be reified and passed into the Pic% base level. Context values are never really used in a Pic% program (though they can be accessed as such through tabulation of a closure or environment). They represent an "evaluation context", which is used frequently in our semantics, as will be explained below.

Dictionaries represent Pic%'s objects. They are built up out of a constant part and a variable part, which are both lists of bindings. The end of a parent-chain or binding-chain is always denoted by **void**. The `clone` operator from section 2.5.3.3 can now be defined as follows:

let $o = dct(cst, var, nxt)$ **in**

$$clone(upTo, o) \equiv \begin{cases} dct(cst, copy(var), nxt) & \text{if } upTo = o \\ dct(cst, copy(var), clone(upTo, nxt)) & \text{otherwise} \end{cases}$$

A.1.2 Evaluation Rules

This section will explain evaluation semantics for each value listed in table A.1. This "interpreter" will be written in an object-oriented style, meaning that we will send an *eval* message to a language value in order to evaluate it. This *eval* message takes exactly one argument (next to its implicit receiver): the evaluation *context*. As noted in table A.1, a context consists of three dictionaries: a "current dictionary" in which evaluation is currently active, a "this dictionary" pointing to the *initial* receiver of the method we are processing and finally a "super dictionary", pointing to the dictionary used in super-sends. We will represent a context as $ctx(c\ \tau\ \sigma)$.

Next to the $eval$ function, defined for each expression $e \in \mathbf{AG}$, the subset of dictionaries also understands a set of accessor and mutator methods. $getCst(nam)$ and $getAny(nam)$ perform dictionary lookup in the constant part and both parts

respectively. $addVar(nam, val)$ and $addCst(nam, val)$ add new bindings to the variable respectively constant lists of an object, while $setVar(nam, val)$ re-assigns a binding's value. We assume these three mutators all return val.

A.1.2.1 Values and Quotations

A large part of Pic%'s first-class object space consists of *values*, which are self-evaluating. Formally, for any $v \in$ **VAL**, we have:

$$v.eval(ctx) \equiv v$$

A quotation allows for the reification of a parse tree. Using this construct, the programmer can directly access parse tree values such as references and definitions. Its evaluation rule is also extremely simple:

$$quo(exp).eval(ctx) \equiv exp$$

A.1.2.2 Definition, Declaration and Assignment

For the evaluation rules of definitions, declarations and assignments, it is best to recall Pic%'s natural 3×3 syntax system which shows each of these language values combines an invocation with some expression. The evaluation rules are thus determined by the *invocation*. For definitions and assignments, they are as follows:

$$\begin{aligned}
def(ref(nam), exp).eval(ctx) &\equiv ctx.c.addVar(nam, exp.eval(ctx)) \\
def(apl(nam, arg), exp).eval(ctx) &\equiv ctx.c.addVar(nam, fun(nam, arg, exp)).wrap(ctx) \\
def(tbl(nam, idx), exp).eval(ctx) &\equiv ctx.c.addVar(nam, maketab(idx.eval(ctx), exp))
\end{aligned}$$

$$\begin{aligned}
ass(ref(nam), exp).eval(ctx) &\equiv ctx.c.setVar(nam, exp.eval(ctx)) \\
ass(apl(nam, arg), exp).eval(ctx) &\equiv ctx.c.setVar(nam, fun(nam, arg, exp)) \\
ass(tbl(nam, idx), exp).eval(ctx) &\equiv ctx.c.getAny(nam).set(idx.eval(ctx), exp.eval(ctx))
\end{aligned}$$

The auxiliary function $maketab(nbr(siz), exp)$, will create a table $tab(a)$ where $a[i] = exp.eval(ctx)$ for $i \in [0, siz[$. The evaluation rules for a declaration are entirely similar to those of a definition, except for the usage of $getCst$ instead of $getVar$. The function $wrap$ will be defined in the next section.

A.1.2.3 Invocations

The evaluation of references, tabulations and applications is what drives a Pic% program. These are frequent operations with important semantics. Most notably,

evaluation of an application will lead to *function application*, which will be explained in more detail in section A.1.2.5. The evaluation rules for references revolve mainly around lookup. Tabulations lead to table indexation.

$$\begin{aligned} ref(nam).eval(ctx) &\equiv ctx.c.getAny(nam).wrap(ctx) \\ apl(exp, arg).eval(ctx) &\equiv exp.eval(ctx).apply(arg, ctx, ctx) \\ tbl(exp, idx).eval(ctx) &\equiv exp.eval(ctx).get(idx.eval(ctx)) \end{aligned}$$

The function $wrap$ is defined below. It is used to "wrap" bare functions in a closure, which "captures" necessary context information to be restored when the function is applied. This is necessary to ensure correct semantics for *first-class methods*.

$$val.wrap(ctx) \equiv \begin{cases} clo(val, ctx) & \text{if } val \in \mathbf{FUN} \\ val & \text{otherwise} \end{cases}$$

A.1.2.4 Qualifications

A qualification can either be a message send or a super send. Their evaluation rules closely resemble those of the above invocations, but their lookup dictionary is specifically qualified. Qualifications can also only be used to query for the *constants* of the qualified dictionary.

$$\begin{aligned} msg(rcv, ref(nam)).eval(e) &\equiv \mathbf{let}\ o = rcv.eval(e)\ \mathbf{in} \\ &\quad o.getCst(nam).wrap(ctx(o\ o\ o.nxt)) \\ msg(rcv, apl(nam, arg)).eval(e) &\equiv \mathbf{let}\ o = rcv.eval(e)\ \mathbf{in} \\ &\quad o.getCst(nam).apply(args, e, ctx(o\ o\ o.nxt)) \\ msg(rcv, tbl(nam, idx)).eval(e) &\equiv rcv.eval(e).getCst(nam).get(idx.eval(e)) \end{aligned}$$

$$\begin{aligned} sup(ref(nam)).eval(e) &\equiv e.\sigma.getCst(nam).wrap(ctx(e.\sigma\ e.\tau\ e.\sigma.nxt)) \\ sup(apl(nam, arg)).eval(e) &\equiv e.\sigma.getCst(nam).apply(args, e, ctx(e.\sigma\ e.\tau\ e.\sigma.nxt)) \\ sup(tbl(nam, idx)).eval(e) &\equiv e.\sigma.getCst(nam).get(idx.eval(e)) \end{aligned}$$

A.1.2.5 Application and Parameter Binding

We have been using the method $apply$ to denote evaluation of applications, message sends and super sends. This application is responsible for both properly evaluating and binding arguments and for evaluating a function body in some new "call frame". Let us start by defining the $apply$ method itself. Apply's semantics differ

depending on the type of the receiver. It receives two context parameters: an actual argument evaluation context acx and a body evaluation context bcx. A closure by itself also contains a closure evaluation context, ccx. Closure application will delegate to function application, where ccx will be used instead of bcx.

$$\begin{aligned} fun(nam, for, bdy).apply(act, acx, bcx) &\equiv \textbf{let } c' = bcx.c.extend().bind(act, for, acx) \textbf{ in} \\ & \quad bdy.eval(ctx(c'\ bcx.\tau\ bcx.\sigma)) \\ clo(fun, ccx).apply(act, acx, bcx) &\equiv fun.apply(act, acx, ccx) \\ nat(nbr(n)).apply(act, acx, bcx) &\equiv \textbf{nativeapply}(n, act, acx, bcx) \end{aligned}$$

The method $extend$ can be simply defined as:

$$rcv.extend() \equiv dct(\textbf{void}, \textbf{void}, rcv)$$

The method simply opens up a new scope to bind the arguments in and to evaluate the function body. Note the subtle differences in context usage of functions and closures. Whereas functions use the "current context" to evaluate both arguments *and* body, closures will only use this context for argument evaluation. The scope and body are related to the closure's encapsulated context. This is necessary to support proper use of first-class methods. It remains to explain the parameter binding mechanism. First, we ensure the actual arguments get evaluated to a table (which is not necessarily the case when using @).

$$\begin{aligned} dct.bind(tab(act), for, e) &\equiv for.call(tab(act), dct, e) \\ dct.bind(exp, for, e) &\equiv for.call(exp.eval(e), dct, e) \end{aligned}$$

Next, the interpreter will dispatch over the type of the *formal* arguments of the function. Canonically defined functions have tables as actual arguments. Functions defined using @ can either have references or applications as formal arguments:

$$\begin{aligned} tab(for).call(tab(act), dct, e) &\equiv for.bindTab(act, dct, e) \\ ref(nam).call(tab(a), dct, e) &\equiv dct.addVar(nam, evalAll(a, e)); dct \\ apl(nam, arg).call(tab(a), dct, e) &\equiv dct.addVar(nam, makeThunks(nam, arg, a, e)); dct \end{aligned}$$

$$\begin{aligned} [\,].bindTab([\,], dct, e) &\equiv dct \\ [for|fs].bindTab([act|as], dct, e) &\equiv for.bindOne(act, dct, e); dct.bindTab(as, fs, e) \end{aligned}$$

We have used Prolog-like syntax to represent the head and tail of the underlying array value of a table. The $evalAll$ auxiliary function takes an array of unevaluated entities and an evaluation context, and returns a table $tab(a')$ where $a'[i] = a[i].eval(e)$. $makeThunks(nam, arg, a, e)$ returns a table $tab(a')$ where $a'[i] = clo(fun(nam, arg, a[i]), e)$. In both cases, it holds that $i \in [0, size(a)[$. In the case of tables, we now need to couple each actual argument to a formal parameter. The semantics depend on the type of formal parameter, to distinguish call-by-value from call-by-name.

$$\begin{aligned} ref(nam).bindOne(act, dct, e) &\equiv dct.addVar(nam, act.eval(e)) \\ apl(nam, arg).bindOne(act, dct, e) &\equiv dct.addVar(nam, clo(fun(nam, arg, act), e)) \end{aligned}$$

A.1.2.6 Capture, this and super Natives

We end an overview of the basic Pic% semantics by defining proper semantics for the natives `capture`, `this` and `super`. Not surprisingly, these natives merely provide access to the invisible evaluation context ever present in the evaluation of an expression. We assume "capture", "this" and "super" to represent the indices of the corresponding natives.

$$\begin{aligned} \mathbf{nativeapply}(capture, tab([\,]), acx, bcx) &\equiv bcx.c \\ \mathbf{nativeapply}(this, tab([\,]), acx, bcx) &\equiv bcx.\tau \\ \mathbf{nativeapply}(super, tab([\,]), acx, bcx) &\equiv bcx.\sigma \end{aligned}$$

A.2 Reintroducing Static Scope

The previous section has defined function application semantics to adhere to *dynamic* scope. This is because functions do not store a lexical environment but rather use one that is passed to them at call-time. Contrast this with closures who do have a paired evaluation context, which they will use when their underlying function is applied. Closures can be used to introduce static scope in Pic%. Indeed, if all functions would be defined as:

```
f := (f(args) : body)
```

Pic% would be a statically scoped language, since `f(args):body` evaluates to a closure and any function is immediately replaced by a closure representing the function's lexical environment. Our method of introducing static scope is thus based on such "closure wrapping", explained in section 5.8.2. Functions will be

wrapped in closures at *method lookup time* rather than at definition time, as shown in the code excerpt above. In concrete, this means that a call to $dct.getCst$ or $dct.getAny$ will *never* return a function anymore. Functions are always wrapped in closures just before they are returned. This means that $apply$ will only have to consider closures and natives. Since closures support static scoping if they are provided with a lexical environment, it remains to be explained how we can provide a function with a proper lexical context at method lookup time.

Remember that we could not store the dictionary of definition inside of a method, because methods should be shared by clones. Care had to be taken to ensure a function is always executed in the right object. The reason for wrapping a function at method lookup time is that the dictionary in which we search for the method is the object in which we should evaluate the method's body.

A function's lexical context – like any context – consists of a "current dictionary", which is the object under which the function's call frame will be hung (i.e. the dictionary that will be extended when the function is applied). The "this" dictionary must refer to the proper *dynamic* receiver of the function, while the "super" dictionary denotes the parent of the *static* (lexical) parent. Thus, `super` is the parent of the "current dictionary". An updated $getCst$ method is defined below which will properly wrap functions in a lexical environment:

$$
\begin{aligned}
dct.getCst(nam, ths) &\equiv dct.cst.lookup(nam, ths, dct) \\
\mathbf{void}.getCst(nam, ths) &\equiv \mathbf{error} \\
bnd(nam, val, nxt).lookup(nme, ths, dct) &\equiv nxt.getCst(nme, ths, dct) \\
bnd(nam, val, nxt).lookup(nam, ths, dct) &\equiv val.wrap(ctx(dct\ ths\ dct.nxt)) \\
\mathbf{void}.lookup(nam, ths, dct) &\equiv dct.nxt.getCst(nam, ths)
\end{aligned}
$$

Any function f found by $lookup$ will be wrapped in a context $ctx(dct\ ths\ dct.nxt)$. Here, dct points to the *static* dictionary in which the function is really *found*. The extra parameter to $getCst$, ths is the dynamic receiver. As explained above, `super` is set to the parent of dct. Evaluating `capture()` inside f will result in a dictionary frm where $frm.nxt = dct$. Evaluating `this()` will result in ths, evaluating `super()` in $dct.nxt$.

The method $getAny$ is updated in an analogous manner. The only other adaptation required to support static scope is to update all calls to $getCst$ and $getAny$ in the above semantics. We will only rewrite those for applications here, to pass the extra ths argument:

$$
\begin{aligned}
apl(nam, arg).eval(e) &\equiv e.c.getAny(nam, e.\tau).apply(arg, e, e) \\
msg(rcv, apl(nam, arg)).eval(e) &\equiv \mathbf{let}\ o = rcv.eval(e)\ \mathbf{in} \\
&\quad o.getCst(nam, o).apply(args, e, ctx(o\ o\ o.nxt))
\end{aligned}
$$

$$sup(apl(nam, arg)).eval(e) \equiv e.\sigma.getCst(nam, e.\tau).apply(args, e, ctx(e.\sigma\ e.\tau\ e.\sigma.nxt))$$

Notice the extra argument to $getAny$ or $getCst$. In super-sends, $e.\tau$ is passed as the dynamic receiver, not $e.\sigma$. This clearly demonstrates "late binding of self": the receiver is left unchanged by super-sends.

A.2.1 Scope Functions

Recall from section 5.4.1 that the natives `this` and `super` can also take an arbitrary expression as an argument, which will be evaluated in the context of `this()` or `super()` respectively. Annotating the evaluation process with contexts, their evaluation is easily expressed:

$$\begin{aligned}\mathbf{nativeapply}(this, tab([exp]), acx, ctx(c\ \tau\ \sigma)) &\equiv exp.eval(ctx(\tau\ \tau\ \tau.nxt))\\ \mathbf{nativeapply}(super, tab([exp]), acx, ctx(c\ \tau\ \sigma)) &\equiv exp.eval(ctx(\sigma\ \tau\ \sigma.nxt))\end{aligned}$$

Late binding of self can again be illustrated by considering the evaluation of `super(this()) = this()`.

A.3 Concurrency Model Semantics

The semantics for cPico are only very briefly discussed since they have only been an intermediate step in the development of dPico. Nevertheless, some important concepts were introduced in cPico which dPico has inherited. This section will briefly outline the necessary language values that will also be needed in the following section.

We assume a function $promise()$ exists which returns a new undetermined promise. Strict operations operating on such promises can use the function $touch(p)$ to access the value. Finally, the function $fulfill(p, v)$ can be used to fulfill a promise with a value. cPico's active objects are represented as a pair $ao(beh, que)$ where $beh \in \mathbf{DCT}$ represents the active object's behaviour and que is a $Queue$. This queue is not first-class, but has two operations enQ and deQ which can be used to enqueue or dequeue evaluation requests. Such an evaluation request is represented as $req(rcv, bdy, ctx, pro)$ where $rcv \in \mathbf{AO}$ is the active object receiving the request, $bdy \in \mathbf{AG}$ is the expression to evaluate, $ctx \in \mathbf{CTX}$ is the context to evaluate bdy in and pro is the promise to fulfill. Such requests are created upon application of a method on an active object. Message sends to active objects are discriminated from messages sent to plain objects when the body context's receiver $bcx.\tau$ is an active object.

$$
\begin{array}{rcl}
\multicolumn{3}{r}{\textbf{let } bcx = ctx(c\ ao(beh, que)\ \sigma) \textbf{ in}} \\
fun(nam, for, bdy).apply(act, acx, bcx) & \equiv & \textbf{let } cf = c.extend().bind(act, for, acx) \textbf{ in} \\
 & & \textbf{let } \rho = promise() \textbf{ in} \\
 & & que.enQ(req(bcx.\tau, bdy, ctx(cf\ beh\ \sigma), \rho)); \\
 & & \rho
\end{array}
$$

The semantics of executing such a request are defined as follows:

$$
\begin{array}{rcl}
ao.exec(req(ao, bdy, ctx, \rho)) & \equiv & lock(ctx.c.nxt); \\
 & & fulfill(\rho, bdy.eval(ctx)); \\
 & & unlock(ctx.c.nxt);
\end{array}
$$

Before execution of the body, it is ensured that the lexical object in which the method was found is properly serialized through the use of the functions $lock$ and $unlock$.

A.4 Distribution Model Semantics

In this section we will briefly try to explain the most important semantics underlying dPico. We will first have to introduce a number of new data types to be able to describe the intended semantics. First, the class of active objects **AO** will have to be redefined. This set is now partitioned in *local* and *remote* active objects, such that $\mathbf{AO} \equiv \mathbf{LAO} \cup \mathbf{RAO}$. Any active object from the previous section is now a local active object $lao(beh, que, nxt)$ consisting of a passive behaviour, a request queue and an added parent pointer.

A remote active object is represented as $rao(loc, id)$ where loc represents the virtual machine on which the remote object resides and id is a unique identification for the remote object, relative to its location. Active closures are represented as $acl(fun, acur, aths)$, where $fun \in \mathbf{FUN}$ denotes a wrapped function, $acur \in \mathbf{AO}$ denotes the implementor of fun, and $aths \in \mathbf{AO}$ denotes the dynamic receiver.

Finally, due to the addition of active object hierarchies, we will extend a context ctx with two new parameters: αc, the lexical active implementor and $\alpha\tau$ the dynamic active receiver, such that

$$
\begin{array}{rcl}
\mathbf{nativeapply}(athis, tab([\,]), acx, ctx(c\ \tau\ \sigma\ \alpha c\ \alpha\tau)) & \equiv & \alpha\tau \\
\mathbf{nativeapply}(asuper, tab([\,]), acx, ctx(c\ \tau\ \sigma\ \alpha c\ \alpha\tau)) & \equiv & \alpha c.nxt
\end{array}
$$

A.4.1 Message Definition

This section more formally explains the semantics of the syntax introduced in section 6.1.3. As noted there, the main purpose of this syntax is to allow a graceful application of higher order functions, which are used to implement special view, mixin and cloning methods. This way, constructing views, active views or cloning methods becomes more visually appealing, while keeping the number of concepts introduced to a minimum. The syntactic notation in Pic% is `f.x:exp`, `f.m(args):exp` and `f.t[i]:exp`. Their semantics are listed below. For declarations, it suffices to replace $addVar$ by $addCst$.

$$def(msg(fun, ref(nam)), exp).eval(e) \equiv \\ e.c.addVar(nam, fun.eval(e).apply(tab([exp.eval(e)]), e, e))$$

$$def(msg(fun, apl(nam, arg)), exp).eval(e) \equiv \\ e.c.addVar(nam, fun.eval(e).apply(tab([fun(nam, arg, exp)]), e, e))$$

$$def(msg(fun, tbl(nam, idx)), exp).eval(e) \equiv \\ e.c.addVar(nam, fun.eval(e).apply(tab([maketab(idx.eval(e), exp)]), e, e))$$

Using these semantics, it becomes clear that we always bind nam to the value of applying some function to a value (in the case of a reference), a function (in the case of an application) or a table (in the case of a tabulation). When using this syntax for applications, it becomes clear that fun must evaluate to some higher order function, since it will take another function as parameter. This is quite special in Pic%, as the function passed as an argument to fun carries a name, yet it is not bound to this name in the environment. Normally, when functions are created in Pic%, they always get bound immediately in the environment. These "message definition semantics" allow for the name of an invocation to be bound to some value different than what regular Pic% semantics prescribe.

A.4.2 Representing Virtual Machines

Since dPico operates across several interpreters, the semantics somehow have to reflect this. We will assume the virtual machine on which an expression is evaluated can be made explicit using the syntax:

$\|exp\|_\pi$

Here, π represents a process or virtual machine evaluating exp. We have already mentioned that remote active objects are represented as $rao(loc, id)$ and that loc represents the virtual machine on which the remote object resides. We will assume the existence of a function $\|resolve(i)\|_\pi$ that resolves the remote active object $rao(\pi, i)$ to a local active object.

Regarding serialization (**m**arshaling) and deserialization (**u**nmarshaling), two functions are introduced. $\mu(v)$ takes any value $v \in \mathbf{AG}$ and marshals it according

to the rules outlined in section 6.2.2. The function $\upsilon(val)$ takes a serialized value and transforms it to a proper dPico value. For most dPico values, $\upsilon(\mu(val))$ results in a copy of val. For active objects, remote references are created. We will introduce some syntactic sugar to ease the description of serializing dPico values:

$$\begin{aligned} \mu_\pi(val) &\equiv \|\mu(val)\|_\pi \\ \upsilon_\pi(val) &\equiv \|\upsilon(val)\|_\pi \\ val_{\pi\to\rho} &\equiv \upsilon_\rho(\mu_\pi(val)) \end{aligned}$$

A.4.3 Active Object Method Invocation

This section describes the semantics behind active object delegation as explained in section 6.5. First of all, $getCst$ will have to be redefined. $getCst$ has taken two arguments up to now: the identifier to be found and the passive dynamic receiver. For active object delegation, $getCst$ is adapted to take as an argument the *active* dynamic receiver. The behaviour of $getCst$ for active objects is then roughly:

let $val = lao.beh.cst.lookup(nam, beh, lao)$ **in**

$$lao.getCst(nam, aths) \equiv \begin{cases} acl(val.fun, lao, aths) & \text{if } val \in \mathbf{CLO} \\ val & \text{if } val \neq null \\ lao.nxt.getCst(nam, aths) & \text{if } val = null \end{cases}$$

$$\|rao(\pi_2, i).getCst(nam, aths)\|_{\pi_1} \equiv resolve(i).getCst(nam_{\pi_1\to\pi_2}, aths_{\pi_1\to\pi_2})_{\pi_2\to\pi_1}$$

Notice how a $getCst$ operation on a remote active object is translated to a $getCst$ operation to a local active object. Since this local object resides on a different VM, the arguments to $getCst$ are serialized and properly unserialized at π_2. The value that was found is then properly serialized and sent back to π_1. Notice that delegation does not involve any promises and thus is purely synchronous. Consider the scenario that an active object closure was found. Application of such a closure happens as follows:

$$\begin{aligned} &acl(fun, acur, aths).apply(arg, acx, bcx) \equiv \\ &\quad \textbf{let } cf = acur.extend().bind(arg, fun.for, acx) \textbf{ in} \\ &\quad\ \textbf{let } \rho = promise() \textbf{ in} \\ &\quad\quad acur.send(cf, fun.bdy, aths, \rho); \rho \end{aligned}$$

$$lao(beh, que, nxt).send(cf, bdy, aths, \rho) \equiv que.enQ(req(cf, bdy, aths, \rho))$$

$$\begin{aligned} &\|rao(\pi_2, i).send(cf, bdy, aths, \rho)\|_{\pi_1} \equiv \\ &\quad resolve(i).send(cf_{\pi_1\to\pi_2}, bdy_{\pi_1\to\pi_2}, aths_{\pi_1\to\pi_2}, \rho_{\pi_1\to\pi_2})_{\pi_2\to\pi_1} \end{aligned}$$

Notice that requests have changed in comparison with cPico. Requests no longer carry a context e since this might contain passive objects which would be copied. Rather, only the call frame cf and the active receiver $aths$ are sent to the active object, together with the body and the promise. How these new requests are handled is shown below. Notice that the call frame's parent is set to the local behaviour. This can only be done only here since it is only at this point that we have the guarantee that the behaviour of the receiving active object is co-located with the call frame. In the code fragment above, cf might have been created on a different VM than the VM where $acur$ resides.

$$
\begin{aligned}
lao(beh, que, nxt).exec(req(cf, bdy, aths, \rho)) \quad \equiv \quad & \textbf{let } e = ctx(cf\ beh\ beh.nxt\ aths\ nxt) \textbf{in} \\
& cf.nxt \leftarrow beh; lock(beh); \\
& fulfill(\rho, bdy.eval(e)) \\
& unlock(beh)
\end{aligned}
$$

A.4.4 Active Object Extension

This section will introduce the semantics necessary to explain how `aview` can create distributed object extensions. We will represent active views as a functor $avw(fun)$. To evaluate the application of such a wrapper function, a new object o is created, whose parent is set to the native dictionary NAT. Next, the arguments are evaluated and bound in this new object. The active view $aview$ itself is a new local active object whose behaviour is o and whose parent is `athis()`. Since `athis()` can be a remote active object, $aview$'s delegation link can be a remote reference. The `aview` method's body is executed in a context where `athis()` will point to the view itself, as can be witnessed by the context parameters. Notice that the final evaluation result is always the active view itself.

$$
\begin{aligned}
avw(fun).apply(arg, acx, bcx) \quad \equiv \quad & \textbf{let } o = \mathrm{NAT}.extend().bind(arg, fun.for, acx) \textbf{in} \\
& \textbf{let } aview = lao(o, new(Queue), bcx.\alpha\tau) \textbf{in} \\
& fun.bdy.eval(o\ o\ o.nxt\ aview\ aview); \\
& aview
\end{aligned}
$$

The view itself is evaluated in the context $ctx(o\ o\ o.nxt\ aview\ aview)$, thus `acurrent` is bound to $aview$. `asuper` will then point to $aview.nxt$, which is $bcx.\alpha\tau$. The `aview` native's behaviour itself can be easily described as:

$$
\mathbf{nativeapply}(aview, tab([exp]), acx, bcx) \quad \equiv \quad avw(exp.eval(bcx))
$$

Where $exp.eval(bcx) \in \mathbf{FUN} \cup \mathbf{CLO}$, otherwise $avw(val)$ is undefined. Similar semantics can be written down for the `amixin` and `cloning` natives.

Appendix B

Examples

B.1 The Same Fringe Problem

The Same Fringe problem is a simple concurrency problem where the elements of two trees are compared to decide whether or not they are equal. Speedup is gained by having two processes generate the fringe of both threes concurrently, and having a third process to do the comparison. To decouple producers (the tree generators) from the consumer (the comparator), a bounded buffer is used to store intermediate results.

The Same Fringe problem is a classical problem which can be used to demonstrate a wide range of concurrency concepts. It requires creation (forking) of concurrency by spawning new processes to handle the generation of the trees. Moreover, since it requires a bounded buffer, it provides for an excellent demonstration of conditional synchronization. Because the bounded buffer is shared between producers and a consumer, the necessary serialization issues such as locking also need to be introduced. The Same Fringe problem is demonstrated using ABCL in (Yonezawa et al., 1986) and PScheme in (Yao and Goldberg, 1994).

Our solution to the problem is shown below. The bounded buffer is expressed using `call-with-current-promise` to achieve conditional synchronization to make a client wait whenever the buffer is empty or full. The Tree generators are active objects who will recursively traverse their tree using an in-order tree walk. Along the way, each element is enqueued in the generator's queue. We need a way to tell the comparator that our tree walk has ended and that no more elements will arrive. This is handled by always passing a "flag" down the tree. The flag will only be true for (direct or indirect) right-hand children of the root. This way, the rightmost leaf is able to detect that it is the last processed leaf, and it can leave behind an "end of tree" token in the queue.

A comparator simply starts up two tree generators and will communicate with them indirectly using a small shared queue. It will then continually dequeue and compare elements from both queues until the token is scanned or two elements differ, in which case it is signalled that both trees do not match. Note that the

bigger the shared queues, the more independent producers and consumer will be able to work. In the case where the queue has size 1, the generators will never be able to get ahead on the comparator by more than one element.

The entire executable example is shown below:

```
{

nil :: [];
null(tree) :: tree=nil;
node(k,f,n) :: [k,f,n];
leaf(v) :: node(v,nil,nil);
key(node) :: node[1];
first(node) :: node[2];
second(node) :: node[3];

screen() :: {
  show@args :: display@args;
  serialize()
};

s :: screen();

queue(siz) :: {
   q[siz]: false;
   ‘in points to next free spot‘
   ‘cnt keeps track of nr of elts in the queue‘
   in: 0; cnt: 0;
   waitforEnQ: void;
   waitforDeQ: void;
   empty() :: cnt=0;
   full() :: cnt=siz;

   enqueue(item) :: {
     if(full(),
        ‘fulfill the promise to enqueue item later‘
        delay(waitforDeQ := [promise,item]));
     if(!is_void(waitforEnQ),
        ‘someone is waiting for an item to fill the queue‘
        { fulfill(waitforEnQ, item);
          waitforEnQ := void },
        { q[in+1] := item;
          in := (in+1)\\siz;
          cnt := cnt+1 });
      item };
```

```
  dequeue() :: {
   if(empty(),
     ‘fulfill the promise to dequeue later‘
     delay(waitforEnQ := promise));

   item : q[((in+siz-cnt)\\siz)+1];
   if(!is_void(waitforDeQ),
     ‘someone is waiting for a dequeue to empty the queue‘
     { q[in+1] := waitforDeQ[2];
      in := (in+1)\\siz;
      fulfill(waitforDeQ[1],waitforDeQ[2]);
      waitforDeQ := void },
     cnt := cnt-1);
   item };

  ‘export interface to ensure enqueue and dequeue‘
  ‘are synchronous operations‘
  interface(me) : {
    enqueue(item) :: touch(me.enqueue(item));
    dequeue() :: touch(me.dequeue());
    capture()
  };

  interface(activate())
};

treeGenerator(q,nam) :: {
   ‘we continually pass a flag to our right child‘
   ‘this makes it possible to identify the rightmost child,‘
   ‘who can then signal the end of the tree‘
   generate(tree, flag) ::
      if(null(tree),
        if(flag, q.enqueue("eot")), ‘signal end of generation‘
        { s.show(nam, " generated: ",
               q.enqueue(key(tree)), eoln);
          activethis().generate(first(tree), false);
          activethis().generate(second(tree), flag) });

   activate()
};

comparator(tree1, tree2) :: {
 q1: queue(5);
```

```
  q2: queue(5);
  g1: treeGenerator(q1,"t1");
  g2: treeGenerator(q2,"t2");

  compare() :: {
     g1.generate(tree1, true);
     g2.generate(tree2, true);
     compareElements(q1.dequeue(), q2.dequeue())
  };

  compareElements(e1, e2) : {
    s.show("comparing ", e1, " and ", e2, eoln);
    if( ((e1="eot") & (e2="eot")), ‘both trees end, stop‘
        true,
        if (e1 = e2, ‘both elements equal, continue comparing‘
           compareElements(q1.dequeue(), q2.dequeue()),
           false))
  };
  activate()
};

test()::{
  exec() :: {
    exTree1:: node(1, node(2, leaf(4), leaf(5) ),
                  node(3, leaf(6), leaf(7) ));
    exTree2:: node(1, node(2, leaf(4), leaf(5) ),
                  node(3, leaf(6), leaf(7) ));
    c: comparator(exTree1, exTree2);
    s.show("both trees equal? ", touch (c.compare()), eoln)
  };
  activate()
};

test().exec();
true
}
```

B.2 A Distributed Chat Client

```
‘ --- At Interpreter A --- ‘
{
‘create an active view on the main active object‘
aview.chatServer(channel, maxClients) :: {
  clients[maxClients] : void;
  occupancy: 0;

  ‘creates an active child of the server‘
  aview.registerClient(nam) :: {
    receiveMsg(from,msg) :: display(from,": ",msg,eoln);
    ‘the child properly registers itself with the server‘
    asuper(
     if (occupancy=maxClients,
        error("Sorry, channel is full"),
        clients[occupancy := occupancy+1] := athis()) )
  };

  ‘broadcast a message to all registered clients‘
  sendMsg(msg) :: {
   from: athis(nam);
   for(i:1, i <= occupancy, i:=i+1,
      clients[i].receiveMsg(from, msg));
   "message sent"
  };

  ‘publish the server to all interested interpreters‘
  register(channel)
};

server: chatServer("vubServer", 10);
tom: server.registerClient("Tom");
tom.sendMsg("Hello world")

}
‘ --- At Interpreter B --- ‘
{
‘get a reference to the chatServer, a remote active object‘
vubserver: members("vubServer")[1];
stijn: vubserver.registerClient("Stijn");
stijn.sendMsg("Hello") }
```

Appendix C

Natives

This appendix gives a brief overview of the crucial natives added to cPico and dPico to support the new concepts of both languages.

Native	Effect or value
cPico	
`activate()`	Creates an active object
`serialize()`	Creates a serialized object
`this()`	Dynamic receiver of a message
`super()`	Parent of lexical object
`activethis()`	Active object currently executing
`this(exp)`	Evaluate exp in the scope of `this()`
`super(exp)`	Evaluate exp in the scope of `super()`
`mixin()`	Mixin current behaviour in the receiver
`cloning(exp)`	Evaluate exp in a clone of the receiver
`delay(exp)`	Allows access to `promise`, interrupts control flow
`fulfill(pro,val)`	Fulfill a promise with a value
dPico	
`this()`	Passive receiver of a message
`super()`	Parent of lexical passive object
`this(exp)`	Evaluate exp synchronously in scope of `this()`
`super(exp)`	Evaluate exp synchronously in scope of `super()`
`athis()`	Active receiver of a message
`asuper()`	Parent of lexical active object
`athis(exp)`	Evaluate exp asynchronously in scope of `athis()`
`asuper(exp)`	Evaluate exp asynchronously in scope of `asuper()`
`view(fun)`	A function able to create a passive view on `this()`
`mixin(fun)`	A function able to create a passive mixin on `this()`
`aview(fun)`	A function able to create an active view on `athis()`
`amixin(fun)`	A function able to create an active mixin on `athis()`
`cloning(fun)`	A function able to create a clone of `this()`
`register(nam)`	Register `athis()` in a channel named nam
`members(nam)`	A table of all registered objects in channel nam
`delay(exp)`	Allows access to `promise`, interrupts control flow
`fulfill(pro,val)`	Fulfill a promise with a value
`copydown()`	Copies all of `asuper()`'s behaviour one level down

Bibliography

Abelson, H. and Sussman, G. J. (1985). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA.

Agha, G. (1986). *Actors: a model of concurrent computation in distributed systems*. MIT Press.

Agha, G. (1990). Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141.

Baker Jr., H. G. and Hewitt, C. (1977). The incremental garbage collection of processes. In *Proceedings of Symposium on AI and Programming Languages*, volume 8 of *ACM Sigplan Notices*, pages 55–59.

Balter, R., Lacourte, S., and Riveill, M. (1994). The Guide language. *The Computer Journal*, 37(6):519–530.

Bardou, D. (1996). Delegation as a sharing relation: Characterization and interpretation.

Bardou, D. and Dony, C. (1996). Split objects: a disciplined use of delegation within objects. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 122–137. ACM Press.

Benton, N., Cardelli, L., and Fournet, C. (2002). Modern concurrency abstractions for C#. In Magnusson, B., editor, *Proceedings of ECOOP02, volume 2374 of LNCS*, pages 415–440. Springer.

Black, A., Hutchinson, N., Jul, E., and Levy, H. (1986). Object structure in the emerald system. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 78–86. ACM Press.

Blashek, G. (1994). *Object-Oriented Programming with Prototypes*. Springer-Verlag.

Borning, A. (1986). Classes versus prototypes in object-oriented languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference, Dallas, Texas, November 1986*, pages 36–40.

Briot, J.-P., Guerraoui, R., and Lohr, K.-P. (1998). Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329.

Briot, J.-P. and Yonezawa, A. (1987). Inheritance and Synchronization in Concurrent OOP. In Bézivin, J., Hullot, J.-M., Cointe, P., and Lieberman, H., editors, *Proceedings of the ECOOP '87 European Conference on Object-oriented Programming*, pages 32–40, Paris, France. Springer Verlag.

Budd, T. (2002). *An Introduction to Object-Oriented Programming*. Addison-Wesley, third edition.

Cardelli, L. (1994). Obliq A language with distributed scope. Technical Report 122.

Cardelli, L. (1998). Abstractions for mobile computation. MSR-TR 34, Microsoft Research.

Caromel, D. (1989). Service, asynchrony and wait-by-necessity. *Journal of Object-Oriented Programming*, 2(4):12–18.

Caromel, D. (1990). Programming Abstractions for Concurrent Programming. In *Technology of Object-Oriented Languages and Systems, PACIFIC (TOOLS PACIFIC '90)*.

Caromel, D. (1993). Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102.

Caromel, D. and Rebuffel, M. (1993). Object based concurrency: Ten language features to achieve reuse. In Ege, R., Singh, M., and Meyer, B., editors, *Proceedings of TOOLS-USA'93, Santa Barbara, (CA), USA*, pages 205–214. Prentice-Hall, Englewood Cliffs (NJ), USA.

Chatterjee, A. (1989). Futures: a mechanism for concurrency among objects. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 562–567. ACM Press.

Codenie, W., D'Hont, K., D'Hondt, T., and Steyaert, P. (1994). Agora: Message passing as a foundation for exploring OO language concepts. *SIGPLAN Notices*, 29(12):48–57.

Connolly, T. and Begg, C. (1999a). *Database Systems: A Practical Approach to Design, Implementation, and Management*. Addison-Wesley, second edition.

Connolly, T. and Begg, C. (1999b). *Database Systems: A Practical Approach to Design, Implementation, and Management*, chapter 19. Transaction Management. Addison-Wesley, second edition.

De Meuter, W. (1998). *Agora: The story of the simplest MOP in the world - or - The Scheme of object orientation*. Springer-Verlag.

De Meuter, W. (2004). *A Prototype-Based Approach to Mobility*. PhD thesis, Vrije Universiteit Brussel. Upcoming.

De Meuter, W., Dedecker, J., and D'Hondt, T. (2003a). Wild abstraction ideas for highly dynamic software.

De Meuter, W., D'Hondt, T., and Dedecker, J. (2003b). Intersecting classes and prototypes. In *Proceedings of PSI-Conference, Novosibirsk, Russia*. Springer-Verlag.

De Meuter, W., Gonzalez, S., and D'Hondt, T. (1999). The design and rationale behind pico.

De Meuter, W., Mens, T., and Steyaert, P. (1996). Agora: reintroducing safety in prototype-based languages.

Decouchant, D., Krakowiak, S., Meysembourg, M., Riveill, M., and de Pina, X. R. (1988). A synchronization mechanism for typed objects in a distributed system. In *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, pages 105–107. ACM Press.

Dedecker, J., Cleenewerck, T., and De Meuter, W. (2003). Distributed object inheritance to structure distributed applications. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*.

Dedecker, J. and De Meuter, W. (2003). Communication abstractions through new language concepts.

Devalez, C. (2003). Application streaming in java. Master's thesis, Vrije Universiteit Brussel.

D'Hondt, T. (1996). The pico programming project. `http://pico.vub.ac.be`.

D'Hondt, T. (2004). Principles of object-oriented languages. `http://prog.vub.ac.be/POOL`.

D'Hondt, T. and De Meuter, W. (2003). On first-class methods and dynamic scope. *Proceedings of LMO*, pages 137–149.

Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457.

Dony, C., Malenfant, J., and Cointe, P. (1992). Prototype-based languages: from a new taxonomy to constructive proposals and their validation. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 201–217. ACM Press.

Ehmety, S., Attali, I., and Caromel, D. (1998). About the automatic continuations in the Eiffel// model. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 219–225.

Feeley, M. (1993). *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University.

Fournet, C. and Gonthier, G. (2002). The join calculus: a language for distributed mobile programming. In *Proceedings of the Applied Semantics Summer School (APPSEM)*, volume 2395, pages 268–332. Springer-Verlag.

Frolund, S. (1992). Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *Proceedings of the Sixth European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag.

Fuggetta, A., Picco, G. P., and Vigna, G. (1998). Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Goldberg, A. and Robson, D. (1989). *Smalltalk-80: The Language*. Addison-Wesley Longman Publishing Co., Inc.

Gosling, J., Joy, B., and Steele, G. (1996). *The Java Language Specification*. GO-TOP Information Inc.

Halstead, Jr., R. H. (1985). Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538.

Hoare, C. A. R. (1973). Hints on programming language design. Technical Report STAN-CS-73-403, Stanford University.

Hoare, C. A. R. (1974). Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557.

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8):666–677.

Hutchinson, N. C., Raj, R. K., Black, A. P., Levy, H. M., and Jul, E. (1991). The emerald programming language. Technical report, Dept. of Computer Science, University of British Columbia, Vancouver, Canada.

Ichbiah, J. D., Barnes, J. G. P., Firth, R. J., and Woodger, M. (1986). *Rationale for the Design of the ADA Programming Language*. The Pentagon, Washington, D. C., 20301, U. S. A., 1986.

ISTAG (2003). Ambient intelligence: from vision to reality. Draft report.

Jul, E., Levy, H., Hutchinson, N., and Black, A. (1988). Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133.

Kafura, D. (1990). Act++: building a concurrent c++ with actors. *Journal on Object Oriented Programming*, 3(1):25–37.

Kafura, D., Mukherji, M., and Lavender, G. (1993). Act++ 2.0: A class library for concurrent programming in c++ using actors. *Journal of Object-Oriented Programming*, 6(6):47–55.

Kafura, D. G. and Lee, K. H. (1989). Inheritance in actor based concurrent object-oriented languages. *Comput. J.*, 32(4):297–304.

Kaminsky, A. and Bischof, H.-P. (2002). Many-to-many invocation: a new object-oriented paradigm for ad hoc collaborative systems.

Lange, D. and Oshima, M. (1998). *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley. `http://aglets.sourceforge.net`.

Lea, D. (1999). *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, second edition. Online Supplement at `http://gee.cs.oswego.edu/dl/cpj`.

Lehrmann Madsen, O., Moller-Pedersen, B., and Nygaard, K. (1993). *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley Publishing Co.

Levy, H. M. and Tempero, E. D. (1991). Modules, objects and distributed programming: issues in rpc and remote object invocation. *Softw. Pract. Exper.*, 21(1):77–90.

Lieberman, H. (1986). 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.

Lieberman, H. (1987). *Concurrent Object-Oriented Programming in Act 1*. MIT Press.

Lieberman, H., Stein, L., and Ungar, D. (1987). Treaty of orlando. In *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 43–44. ACM Press.

Liskov, B. (1988). Distributed programming in argus. *Communications Of The ACM*, 31(3):300–312.

Liskov, B. and Shrira, L. (1988). 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.

Lucas, C. and Steyaert, P. (1994). Modular inheritance of objects through mixin-methods. In *Proceedings of the 1994 Joint Modular Languages Conference*, pages 273–282.

Maheshwari, U. and Liskov, B. (1995). Collecting cyclic distributed garbage by controlled migration. In *Proceedings of PODC'95 Principles of Distributed Computing*.

Malenfant, J., Dony, C., and Cointe, P. (1992). Behavioral Reflection in a prototype-based language. In Yonezawa, A. and Smith, B., editors, *Proceedings of Int'l Workshop on Reflection and Meta-Level Architectures*, pages 143–153, Tokyo.

Matsuoka, S. (1993). *Language Features for Re-use and Extensibility in Concurrent Object-Oriented Programming*. PhD thesis, The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan.

Matsuoka, S. and Yonezawa, A. (1993). Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research directions in concurrent object-oriented programming*, pages 107–150. MIT Press.

Meyer, B. (1993). Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80.

Milicia, G. and Sassone, V. (2004). The inheritance anomaly: ten years after. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1267–1274. ACM Press.

Milner, R. (1993). The polyadic pi-calculus: a tutorial. In Bauer, F. L., Brauer, W., and Schwichtenberg, H., editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag.

Mulet, P. and Cointe, P. (1993). Definition of a reflective kernel for a prototype-based language. In Nishio, S. and Yonezawa, A., editors, *Proceedings of the 1st JSSST International Symposium on Object Technologies for Advanced Software, Kanazawa, Japan*, pages 128–144. Springer-Verlag, Berlin.

Norcross, S. J. (2003). *Deriving Distributed Garbage Collectors from Distributed Termination Algorithms*. PhD thesis, University of Saint Andrews.

Open E Project (2004). E: Open Source Distributed Capabilities. `http://www.erights.org`.

Philippsen, M. and Haumacher, B. (1999). More efficient object serialization. In *IPPS/SPDP Workshops*, pages 718–732.

Piquer, J. M. (1991). Indirect reference counting: A distributed garbage collection algorithm. In Aarts, E. and van Leeuwen, J., editors, *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE'91), Eindhoven, The Netherlands*, volume 505 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany.

Pratikakis, P., Spacco, J., and Hicks, M. (2003). Transparent proxies for java futures.

Raj, R. K., Tempero, E. D., Levy, H. M., Black, A. P., Hutchinson, N. C., and Jul, E. (1991). Emerald: A general-purpose programming language. *Software - Practice and Experience*, 21(1):91–118.

Schougaard, K. (2003). Language support for distributed computation.

Smith, R. B. and Ungar, D. (1995). Programming as an experience: The inspiration for self. *Lecture Notes in Computer Science*, 952:303–??

Smith, W. R. (1995). Using a prototype-based language for user interface: the newton project's experience. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 61–72. ACM Press.

Steyaert, P. (1994). *Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks*. PhD thesis, Vrije Universiteit Brussel.

Steyaert, P., Codenie, W., D'hondt, T., De Hondt, K., Lucas, C., and Van Limberghen, M. (1993). Nested mixin-methods in agora. *Lecture Notes in Computer Science*, 707:197–??

Steyaert, P. and De Meuter, W. (1995). A marriage of class- and object-based inheritance without unwanted children. In *Proceedings of ECOOP '95*, volume 952 of *Lecture Notes in Computer Science*, pages 127–144. Springer.

Stiegler, M. (2000). The E language in a walnut. `http://www.skyhunter.com/marcs/ewalnut.html`.

Stroustrup, B. (1986). *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc.

Sun Microsystems (2004). Java 2 Micro Edition. `http://java.sun.com/j2me`.

Surribas, E., Oktaba, H., and Huerta, E. (1996). cc++: a concurrent object-oriented language. *Revista de la Sociedad Chilena de Ciencia de la Computacion*, 1(1):15–30.

Taivalsaari, A. (1993). *A Critical View of Inheritance and Reusability in Object-oriented Programming*. PhD thesis, University of Jyvaskyla.

Taivalsaari, A. (1996). Classes vs. prototypes - some philosophical and historical observations.

Taura, K., Matsuoka, S., and Yonezawa, A. (1994). Abcl/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation. In *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms,1994*.

Thorn, T. (1997). Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239.

Tolksdorf, R. and Knubben, K. (2001). dself - a distributed self. KIT-Report 144, TU Berlin.

Tolksdorf, R. and Knubben, K. (2002). Programming distributed systems with the delegation-based object-oriented language dself. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 927–931. ACM Press.

Ungar, D., Chambers, C., Chang, B.-W., and Hölzle, U. (1991). Organizing programs without classes. *Lisp Symb. Comput.*, 4(3):223–242.

Ungar, D. and Smith, R. B. (1987). Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242. ACM Press.

Van Belle, W. and D'Hondt, T. (2000). Agent mobility and reification of computational state, an experiment in migration. In *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, number 1887 in Lecture Notes in Artifical Intelligence. Springer Verlag.

Van Belle, W. and Fabry, J. (2001). Experiences in mobile computing: The cborg mobile multi agent system. Presented at Tools Europe 2001.

Van Belle, W., Verelst, K., Fabry, J., and D'Hondt, T. (2000). The cborg mobile multi-agent system. `http://cborg.sourceforge.net`.

Van Belle, W., Verelst, K., Van Buggenhout, K., and D'Hondt, T. (2001). Is message sending good enough? communication and synchronisation revisited. Accepted at ECOOP 2001, Workshop 17.

Van Cutsem, T., Mostinckx, S., De Meuter, W., Dedecker, J., and D'Hondt, T. (2004). On the performance of soap in a non-trivial peer-to-peer experiment. In *Proceedings of the 2nd International Working Conference on Component Deployment*, Lecture Notes In Computer Science. Springer Verlag.

Verelst, K. and Van Belle, W. (2000). Synchronization and communication in mobile multi agent systems csp revisited. Submitted at ECOOP 2000, Workshop 7.

Vitek, J., Serrano, M., and Thanos, D. (1997). Security and communication in mobile object systems.

Weiser, M. (1991). The computer for the twenty-first century. *Scientific American*, pages 94–100.

Wilson, P. R. (1992). Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France). Springer-Verlag.

World Wide Web Consortium (2003). Simple object access protocol (soap) 1.2 w3c note. `http://www.w3.org/TR/SOAP/`.

Yao, C. and Goldberg, B. (1994). Pscheme: Extending continuations to express control and synchronization in a parallel LISP. Technical Report TR1994-655, New York University.

Yonezawa, A., Briot, J.-P., and Shibayama, E. (1986). Object-oriented concurrent programming in abcl/1. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268. ACM Press.

Index