KTH Information and Communication Technology

The Design Philosophy of Distributed Programming Systems: the Mozart Experience

Per Brand

A dissertation submitted to
the Royal Institute of Technology
in partial fulfillment of the requirements for
the degree of Doctor of Philosophy

June 2005

The Royal Institute of Technology
School of Information and Communication Technology
Department of Electronics and Computer Systems
Stockholm, Sweden

TRITA-IMIT-LECS AVH 05:04
ISSN 1651-4076
ISRN KTH/IMIT/LECS/AVH-05/04–SE

and
ISRN SICS-D–37–SE
SICS Dissertation Series 37
ISSN 1101-1335

Abstract

Distributed programming is usually considered both difficult and inherently different from concurrent centralized programming. It is thought that the distributed programming systems that we ultimately deploy, in the future, when we've worked out all the details, will require a very different programming model and will even need to be evaluated by new criteria.

The Mozart Programming System, described in this thesis, demonstrates that this need not be the case. It is shown that, with a good system design, distributed programming can be seen as an extended form of concurrent programming. This is from the programmer's point-of-view; under the hood the design and implementation will necessarily be more complex. We relate the Mozart system with the classical transparencies of distributed systems. We show that some of these are inherently on the application level, while as Mozart demonstrates, others can and should be dealt with on the language/system level.

The extensions to the programming model, given the right concurrent programming base, are mainly concerned with non-functional properties of programs. The models and tuning facilities for failure and performance need to take latency, bandwidth, and partial failure into account. Other than that there need not be any difference between concurrent programming and distributed programming.

The Mozart Programming System is based on the concurrent programming language Oz, which integrates, in a coherent way, all three known concurrency or thread-interaction models. These are message-passing (like Erlang), shared objects (like Java with threads) and shared data-flow variables. The Mozart design philosophy is thus applicable over the entire range of concurrent programming languages/systems. We have extracted from the experience with Mozart a number of principles and properties that are applicable to the design and implementation of all (general-purpose) distributed programming systems.

The full range of the design and implementation issues behind Mozart are presented. This includes a description of the consistency protocols that make transparency possible for the full language, including distributed objects and distributed data-flow variables.

Mozart is extensively compared with other approaches to distributed programming, in general, and to other language-based distributed programming systems, in particular.

Acknowlegements

First and foremost I would like to thank the entire Mozart development group. It was a priviledge and pleasure to work with such a talented, creative and strong-willed group of people. The development of Mozart was very much team work, and the number of people involved in Mozart, at one time or other, was very large for an academic project.

It was, I always thought, appropriate that the distributed programming system Mozart was also developed in a distributed fashion with people from SICS/KTH in Sweden DKFI/Saarland University in Germany and later UCL in Belgium. I would like to thank the people who made my frequent and lengthy visits to Saarbrucken during the early Mozart days so pleasant and rewarding - Ralf Scheidhauer, Christian Schulte, Martin Henz, Konstantin Popov, Michael Mehl and Martin Muller (and his wonderful wife Bettina).

I especially want to thank Seif Haridi, my advisor, and Peter van Roy for their encouragement, friendship and insights. I will always remember, fondly, the intense but friendly discussion atmosphere that we had during the course of the Mozart project (and thereafter).

I would also like to thank Gert Smolka's group at DKFI for the development of early Oz. This gave us a good base without which Mozart would never have been realized.

I want to thank Vinnova(Nutek) for their support, especially in the early days when the ground work for Mozart was laid in the Perdio project.

Finally, I want to thank Sverker Janson, Erik Klintskog and Ali Ghodsi for their valuable comments on the drafts of this document.

Contents

Chapter 1

Introduction

This dissertation presents the Mozart Programming System. Design and implementation issues are covered and the broader implications of the work are extensively discussed.

Mozart is a general-purpose distributed programming system, a system designed specifically for the programming of distributed applications. Like all programming systems such a system needs to be understood and evaluated in view of its fundamental purpose: to enable and simplify the development of applications. In this case, the applications that we are targeting are distributed, i.e. intended to run on more than one machine.

Mozart is a complete distributed programming system. Released in 2000 it has been extensively tested and proven in practice. It is self-contained, it contains all that is needed to develop most distributed applications. (It does, of course, like all programming systems, make use of the standard operating system services in Unix and Windows).

We take the position that a distributed programming system is a realization of a distributed programming language. This is not always the way in which distributed programming systems are viewed. Often, the tools by which distributed applications are developed are thought of consisting of a centralized programming system augmented by a number of libraries for distribution, but this poorly reflects the challenges for the application programmer when moving from centralized to distributed applications. The distribution libraries would, unlike typical libraries for centralized programming, be part of the core of the system, and not an optional, occasionally used, add-on.

Put another way, whatever people might choose to call the packages of tools that promote for the purpose of developing distributed applications, there are a number of necessary properties that such tool packages must have to be at all useful. Programmers need to be provided (at least in order to avoid laborious trial-and-error programming) with a precise model of the functional properties (semantics) of the available programming constructs and some, though possibly less precise, model over various non-functional properties (like performance, and sometimes failure and security). The former, the semantics, is, of course, just what you expect to find in a programming language and the latter, the non-functional properties, in a programming system. These

packages of tools can thus be considered distributed programming systems.

Mozart is thus one particular distributed programming system, a realization of one particular distributed programming language. Mozart is extensively compared with alternatives. Factors such as expressivity (normally thought of as a language property) and performance (a system quality) are considered. We will argue that Mozart is a powerful distributed programming system with unparalleled expressivity, and an easy to understand performance and performance-tuning model.

The core of this thesis consists of four longer papers, three journal papers and one unpublished paper of journal length which is an extended version of a published conference paper. The first of these papers focuses on language issues and the programming model. Both functional and non-functional aspects of the programming system are covered.

The focus of the other three papers is on how Mozart was realized. In order to realize Mozart, we faced a large number of design and implementation challenges. Chief among these challenges were the development of suitable protocols (or distributed algorithms) to support various kinds of language entities (e.g. objects, procedures, and immutable data structures) that are shared between sites (i.e machines).

1.1 The Mozart Experience

We also discuss the broader implications of the work and attempt to systematically place the Mozart work into a wider context. One of the important reasons for doing so is that distributed programming subsumes centralized programming. This means that Mozart and all other distributed programming systems also commit the programmer to a given centralized programming language (Oz in the case of Mozart). But the virtues, or lack thereof, of the various centralized programming systems has been debated for 40 years and no consensus has yet been reached. As this thesis is exclusively concerned with distribution we will try to sidestep this issue as much as possible.

The Mozart experience, the principles that we formulated, and the insights that we gained have wide applicability. As Mozart/Oz caters for all the major programming paradigms (functional, object-oriented, and data-flow) the principles are applicable to the design of any distributed programming language/system based on any (or any combination) of these paradigms.

We will consider the question, how should one go about developing a distributed programming system in general? What are the different approaches? We argue that there are currently only two. All the more interesting and more expressive distributed programming systems, including Mozart, belong to one category. In this category distribution support is to some extent integrated into a concurrent centralized programming language/system. In the centralized concurrent system threads (or processes) share language entities according to an entity-specific model (e.g. shared objects, shared code, shared data). This we call the *sharing model*. Integration is achieved by supporting - once again, to a certain extent - the same sharing model between sites (across the net). There are obvious advantages to having the same (or even similar)

sharing model between sites as within a site. It makes for a simpler programming model; concurrent programming is naturally extended to distributed programming. Alternatively, inverting the relationship, a good distributed programming model will subsume a good concurrent programming model.

We shall see that the most important characteristic of the sharing model, from the point-of-view of distribution support, is those aspects that allow threads (processes) located at different machines to interact. Without interaction the distributed system is trivial and once initialized each site works independently and in isolation. So whether we are considering code or data, the important consideration is the dynamic aspects of the sharing model - those that allow additional code or data to be shared. When we analyze concurrent programming languages from this perspective we find that there are only three different models in all existing programming languages.

From this point-of-view distributed programming systems can be evaluated by considering how well distribution support is integrated into the programming language. Having embarked on the path of making sharing between sites much like sharing on one site, there needs to be a good reason not to make them completely identical or similar. There are two potential reasons. First, it may not be possible, and second it may not be practical.

There are conceptual limits to integration, but we show that it is possible to make the distribution sharing model very similar to the concurrent sharing model. This requires, among other things, a rich and expressive concurrent sharing model. The differences between the distribution and concurrent sharing models can then be limited to certain non-functional aspects. Furthermore these non-functional aspects can be dealt with orthogonally to program functionality.

The practical limits to integration, both supposed and real, are discussed extensively. We shall see, within each of the three sharing paradigms, that there are practical limitations associated with many concurrent programming languages. However, we shall see that most of these limitations are not truly conceptual. Rather, the language was not designed with distribution in mind, and the language lacks some functionality or expressiveness that was deemed not essential or overlooked in the centralized scenario. Irrespective of whether this was a correct or incorrect choice in the centralized case, these deficiencies must be attended to in the distributed case. Examples of this are languages where only data-sharing is explicit in the language (i.e. code is shared implicitly by name), and where the distinction between mutable and immutable data is blurred.

How does Mozart fit into this? Mozart is based on a concurrent programming language that contains all three sharing models. Within each paradigm the system is maximally integrated and hence is easily compared to other systems. It is shown that other systems could be better integrated than they are. In some cases this reflects a major lack of distribution support, in other cases it is matter of extending the programming language.

Finally, we consider the question of whether having three sharing models within one language is really necessary. First the arguments for three sharing paradigms in a concurrent but centralized setting are reviewed. The concurrent programming language

Oz, upon which Mozart was based, supported all three sharing models long before distribution was considered. We then show that when distribution is added that the arguments for the usefulness of providing for the entire spectrum of sharing models is much stronger.

We conclude, therefore, that the Mozart system is a useful and powerful tool for building distributed applications. Furthermore, the Mozart work demonstrates a number of design principles and philosophies that should be used in the development of all general-purpose distributed programming systems.

1.2 Overview

This thesis is organized into three parts. Each part starts with its own overview chapter.

The first part briefly summarizes the work and puts the four included papers into context. Also, as the Mozart system was joint work, the specific contributions made by the author are carefully described. Finally a number of additional design and implementation issues that we faced are briefly described.

After a short introduction the second part consists of the four included papers, three of which are journal papers, while the fourth is an unpublished longer version of a published conference paper.

The third part discusses the wider implications of the work. Here we begin by going back to basics and consider the question of what makes a good programming system in general before moving on to distributed programming systems. We formulate criteria to evaluate distributed programming systems and apply them to Mozart and other systems. We consider the question of what a distributed programming language should contain irrespective of user preferences for centralized programming languages/systems. We finish by describing future work (much of which has been initiated today).

1.3 Reading recommendations

Depending on the interests of the reader this thesis can be read in a number of different ways.

The first half of part I and the first of the four papers (chapter 6 in part II)focus on Mozart, as a the distributed programming language, and can be read independently of the rest of the thesis.

The second half of part I and papers 2-4 (chapters 7, 8, and 9 in part II) focus on the implementation design and protocol support.

Also each of the four papers in part II is more or less self-contained and can be read separately.

Finally, part III is self-contained and is a general formulation of the principles that should be used in the design of distributed programming languages/systems. The work on Mozart is used to support that position. Mozart demonstrates many of the

characteristics that good distributed programming systems should have, and, as we shall show, comes closer to fulfilling the criteria than other systems.

Finally, for the reader not familiar with Mozart or Oz as a programming language only fairly short summaries are provided in this thesis. The interested reader can also download the Oz tutorial at http:\\www.mozart-oz.org [94]. Finally the book 'Concepts, techniques, and models of computer programming' [129], is the most comprehensive expose of the Oz programming language (there is even a chapter on distributed programming).

Part I

The Mozart Programming System

Overview of Part I

This part consists of four chapters. In the first chapter we focus on the language aspects of the Mozart system. We relate the role of a distributed programming language, in general, and Mozart, in particular, to the classic distributed system goal of transparency. This chapter is supported by the first of the four papers (chapter 6).

In the second chapter we focus on the system aspects of Mozart. If the first chapter is the *what*, then this chapter is the *how*. A central role is played by the protocols that coordinate the language entities that are shared between sites. This chapter is supported by papers 2-4 (chapters 7,8 and 9) which are all devoted exclusively to the more complex of the protocols. In addition, we also summarize a number of other design and implementation issues that arose during the course of realizing the Mozart system.

In the third chapter we compare Mozart with other state-of-the-art distributed programming systems.

In the fourth and final chapter of this part of the thesis the particular contributions by the author to the joint work are described.

Chapter 2

Distributed Programming Languages

When we use the term *distributed programming system* we mean the complete set of tools by which distributed applications can be programmed. Not included are the proper libraries which are there for convenience. Proper libraries are software components that in turn were developed within the distributed programming system, but that have been found useful enough to be put in some repository for future use.

The term *distributed programming language* then refers to the language by which the programmer interacts with and instructs the distributed programming system. Programming languages have semantics (e.g. operational) defining the behavior of the primitive programming constructs. Non-functional properties of a system (like performance) are system qualities, reflecting that the same programming language may be realized (as programming systems) in better or worse ways.

The only reason that we might be belaboring the point about libraries is to lay the groundwork for a fair comparison. One method, all too often used, to hide complexity of programming languages/systems is to present part of the language as libraries. However, if the libraries are native (i.e. not expressible in the language) and at the same time used over a wide range of applications, then the programmer must also have a good understanding, both as regards functional and non-functional aspects, of these 'libraries'.

In this section we consider the question of what makes for a good and useful distributed programming language. We relate the language question to the traditional goal of transparency in distributed systems. We see that taking the language point-of-view actually helps to bring some order in the multitude of transparencies that the distributed system community defines. We define network-transparency (from the language point-of-view) and argue for its usefulness as demonstrated by Mozart.

The limits of transparency are also discussed. Total transparency is not always possible and sometimes it is not desirable. We show that in Mozart fundamental limitations of transparency do not detract from the usefulness of network transparency. We also show that Mozart respects the limits of good transparency (i.e. not offering more than is desirable).

We discuss the relationship between concurrency and distribution. We briefly summarize the Mozart/Oz computation model and show that the well-designed concurren-

cy model of Mozart also simplifies distribution. Finally, we briefly discuss language extensions for distribution that have no concurrent programming counterpart and exemplify with the Mozart provisions for open computing.

2.1 Transparency

In almost any textbook on distributed systems the merits of transparency are clearly formulated. For instance, in Tanenbaum & van Steen [122], it says 'an important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers'. This goal also forms the definition of transparency, 'a distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be transparent'.

This seems clear enough and most users can easily recognize its validity on the application level. Users do not need to know where a specific URL is located to browse a web page, and both the user and the server source can be physically moved and transparency means that the user notices no difference (at least ideally, but we will get back to the limits of transparency).

In Tanenbaum & van Steen, very early in the book, a number of different types of transparency are listed. They list:

access transparency: Hides differences in data representation and how a resource is accessed

location transparency: Hides where a resource is located

migration transparency: Hides that a resource may move to another location

relocation transparency: Hides that resource may be moved during use

replication transparency: Hides that a resource is replicated

concurrency transparency: Hides that a resource may be shared by several users

failure transparency: Hides the failure and recovery of a resource

persistence transparency: Hides whether a software resource is in memory of on disk

We note that there seems to be many different kinds of transparencies. A number of questions come to mind. First, is this list complete? Second, why are there so many? Third, are they all on the same level? Finally, can the various transparencies be ordered in some way?

The answer to the first question is no. Looking at another textbook on distributed systems Coulouris, Dollimore and Kindberg [29], we find in addition:

mobility transparency: allows the movement of clients within the system

performance transparency: allows the system to be reconfigured to improve performance as loads vary

scaling transparency: allows the system to expand in scale without change to system

With additional effort we could undoubtedly find many more transparencies.

The answer to the second question is partly that the different types of transparency reflect transparency as seen by the user. It is transparency on the application level. The number of applications is unbounded, and even if we attempt to classify applications, the number of different types of applications is very large and multi-faceted. For instance the difference between mobility transparency and location transparency is that in the one case the user moves and in the other the resource being used is moved. Relocation transparency is also related, reflecting as it does movement during the running of the application rather than at startup.

Furthermore, transparency also begins to include artifacts - in the sense that there are mechanisms used in distributed systems to improve fault-tolerance and/or performance and then transparency reflects that the improvements only improve, i.e. have no untransparent side-effects. An example is replication transparency. Replication is done to improve performance and/or fault-tolerance.

Finally, and this may be natural, reflecting the community's consensus that transparency is a good thing, some of the transparencies seem to go beyond the definition. Desirable qualities, yes, but not true transparency. For instance, scaling transparency is not a characteristic of 'as it were only a single computer system'. Single computers are all limited in capacity.

2.2 Network-transparent language

2.2.1 Ordering transparencies

We will now answer the fourth question posed in the previous section and impose some order into the multitude of transparencies. We no longer look at transparency from the user or application perspective, but rather from the programming language perspective. While there are are a multitude of applications and application types, with various properties, there exist only a limited number of programming languages and programming language types. The space of possibilities is much smaller. Also, conceptually, for most applications only a single general-purpose programming language/system is actually needed (while we do need many different types of applications).

A maximally transparent distributed programming language then lets the distributed application programmer program, in so far as possible, his application 'as if it were a single computer system'. This is described most fully in paper 1 (chapter 6) and called there *network transparency*. On the language level network transparency means that the semantics is independent of how the application is distributed among a set of machines. This includes the special case when all threads/process run on the same machine.

Looking at the various types of transparencies, we can now order them. First access transparency, location transparency, migration transparency and mobility transparency, are all part of network transparency. Mozart exhibits them in full. For example, an object (both code/methods and state) is a software resource. It may be accessed/used by any site that has a reference to it (and differences in data representation are hidden).

What are called scaling transparency and performance transparency belong on the application level or tool/library level. Not all applications scale. Of course, there are distributed programming language/system qualities that can make scaling applications harder or easier. This is important, as discussed in the next subsection and section 2.3.

2.2.2 Concurrency

Reformulating the definition of transparency on the language level we get - 'a distributed programming system that is able to present itself to programmers as if it were only a single computer system is said to be network-transparent'.

It is clear from this definition that the language must, to be general-purpose, concurrent. Without concurrency a distributed application would consist of only a single virtual execution process that would pass from machine to machine like a token. Usually execution takes place on many machines concurrently (i.e. at the same time).

From the programming language point-of-view the details of concurrency need to be carefully worked out before distribution is even considered. In the case of Mozart the concurrent core was based on the concurrent programming language Oz [33], developed earlier and conservatively extended for distribution. If concurrency is first introduced in conjunction with distributing an application then it might seem to be connected to distribution, but it is not.

Concurrency is fraught with a number of programming traps that do not occur in non-concurrent (and centralized) programming. Simultaneous access and updates to mutable state can lead to inconsistent state. In order to avoid this the programmer must synchronize between the concurrent processes/threads, via locks (or derivatives thereof, like monitors) or transactions. In general, the programmer must carefully avoid the pitfalls of oversynchronization (e.g. deadlock) on the one hand, and undersynchronization (e.g race conditions) on the other. Concurrency transparency just means that the pitfalls are successfully avoided. Once again, this is on the application level.

However, the properties of the language and system are still important and can aid or hinder the programmer in dealing with concurrency. On the programming level, the distributed and concurrent programming language/system should give the programmer all the tools needed to avoid the pitfalls of concurrency. Mozart/Oz does this. It provides the programmer (in addition to locks) a weaker but safer mechanism for synchronization, the single-assignment or data-flow variable. This is safer in that this is deadlock free, but weaker in the sense that they are not enough for all applications (but very useful for many). This programming technique, declarative concurrency, is presented in [129].

Good concurrency mechanisms can also help in achieving, on the application level, scaling transparency or, more generally, better scaling. The key to scaling, when this

cannot be done on the algorithmic level, is to put more machines to work on the problem. For this to work well the application must have two properties. First, the application needs to be (or can be made to be, by clever parallelization) massively concurrent. Second the dependencies between the concurrent agents must be limited. Applications where the dependencies are almost non-existent have been called *embarrassingly parallel*. If the dependencies are too large, the computational cycles or memory that additional machines provide will not compensate for the increased synchronization between threads/processes on different machines. The more machines that are added the more the threads/processes are partitioned and the more the synchronization takes place between threads/processes on different machines, which is much slower due to network latency. There comes a point where no gain in performance will be had by adding more machines - some limit is reached. Good concurrency mechanisms can push that limit quite a bit further. This is demonstrated in [102] [103].

2.2.3 The Language View

Replication transparency is also, in a sense, part of network transparency. From the language point-of-view, the distributed programming system is free to replicate in order to improve performance (or fault-tolerance) as long as this is safe, i.e. does not change language semantics.

We defer considering failure transparency (or its cousin persistence transparency). However as for the other transparencies we see that they form two groups. One group can and should be realized on the language level, and the other which is on the application level, while the other is largely beyond the scope of this thesis.

Ultimately we are interested in applications. But the properties of network- transparency *carry over* from the language/system to the application. The virtue of a network-transparent distributed programming language/system is that it easy to write distributed applications that exhibit access transparency, location transparency, etc. Some, but not all, of the desired transparencies on the application level are obtained virtually for free.

In paper 1 (chapter 6) we present a distributed graphical editor application. The application was developed (i.e programmed and tested) on a single computer and then trivially extended to distribution.

2.2.4 Summary

Our argument as to the relationship between the traditional goals of transparency in distributed systems to language network-transparency as demonstrated by Mozart is summarized in the three step argument below. We keep in mind the axiom that 'an important goal of a distributed system is to hide the fact that its processes are physically distributed across multiple computers'.

From Tanenbaum & van Steen: A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be transparent.

Our Language View: A distributed programming language that is able to present itself to programmers as if it were only a single computer system is said to be network-transparent.

Our System View: A distributed programming system that allows programmers to develop and test their distributed applications on a single computer system is practically network-transparent.

2.3 Limits of Transparency

In [122] the limits of transparency are also discussed. They are:

Timings and performance: Different distribution structures (the particular partitioning of the application over a set of sites) of the same application may impact timings and performance considerably due to varying physical or logical network distances between machines.

Location-aware applications: Clearly there are some applications where it is not desirable to hide location of the user, as the application need this information to filter and adapt the application accordingly.

Failure: Failure transparency is not always achievable, and is not always desirable.

In the above descriptions there are two very different factors that are mixed. First, complete transparency is just not possible. Second, transparency is not always desirable.

From the distributed language point-of-view one is, of course, forced to accept the fact that complete transparency is impossible. An important part of network awareness, as described in paper 1 (chapter 6), is to provide the programmer with a practical model to deal with this. Note that in paper 1 the term *network awareness* is general and covers both this awareness aspect as well as control aspects (e.g. being able to control where computations run).

One important aspect when considering timings or performance, is the number of network hops that the various language operations will take. In Mozart it is shown that worst case depends only on the language entity type (e.g. an object), and the expected case depends on the usage pattern.

There is nothing unusual about this. The point here is that you do not lose awareness due to transparency. Simpler distributed programming systems/tools (with poor network-transparency) have similar models. For instance, consider RMI (remote method invocation) [120] or RPC (remote procedure call). Worst case here is two network hops, but expected case can differ (when the remote process actually resides on the same machine). One of the main design criteria that we used in pursuit of network transparency (up to the impossibility limits) was not to lose the awareness that simpler and less transparent systems invariably have.

Location awareness or lack thereof is, however, an application question. The fact that Mozart is network transparent does not preclude applications from reasoning about and adapting to location.

2.4 Partial Failure

2.4.1 Failure Transparency

Previously we covered the first two non-transparencies, timings and location-awareness. We now turn our attention to failure.

Distributed systems unlike centralized systems exhibit partial failure, e.g. one machine out of a set of machines involved in the same application fails. Failure transparency, is defined in Tanenbaum as 'the user does not notice that a resource (he has possibly never heard of) fails and the system subsequently recovers from that failure'. Notice that the definition clearly puts failure transparency on the application or user level.

Furthermore, Tanenbaum goes on to state that there is a trade-off between a high degree of transparency and the performance of a system. Here he is thinking of two very different issues.

The first issue is that various forms of relaxed consistency can on the application level be thought of as a lack of transparency. Many replication schemes introduce what in the context of centralized programming would be considered inconsistency (i.e. not sequentially consistent). From the language point-of-view this is a question of semantics. Mutable sequential consistent state and mutable, say, eventually consistent state are two different semantic types of entities. Both have their uses, and distributed programming systems should support (directly as primitives or indirectly as libraries) both. The best choice is application dependent and the network awareness model is one of the factors used to decide which is appropriate.

The second issue is that failure transparency (where possible) is very expensive. On the user and application level it is definitely not something you always want - sometimes it is better to give up.

Even when you do want failure-transparency on the user level this does not translate into a practical goal on the language and programming system level. Fault-tolerance may be achieved on many different levels of granularity. On a very fine-grained level, fault-tolerant techniques making use of redundancy may be used to be able to recover from all crash failures on the level of individual memory cell updates. No single object will ever be left in a inconsistent state. This can be done (given reliable failure detection) but is enormously costly. It may be that fault-tolerance can be achieved on a coarser level, throwing away intermediate results, and restarting from an earlier point. This coarse grained fault-tolerance can cause very long delays when failures do occur (particularly, upon repeated failures), but cost little when failures do not occur. The dependencies between mechanisms for fault-tolerance and system performance indicate that fault-tolerance is on the application level and not on the language level. The

appropriate trade-off choice is application dependent.

2.4.2 The Mozart Failure Model

Nevertheless, dealing with failure was an important aspect of the Mozart work. We needed to give the programmer the means to avoid creating distributed applications that suffer from the syndrome so succinctly described by Leslie Lamport 'you know you are dealing with a distributed system when the crash of computer you never heard about stops you from getting any work done'.

There was a real danger here. Network transparency hides the identities of sites from one another. Clearly the information that a site with a given IP-address and port has crashed breaks transparency. Not only that, but it would be difficult for the programmer to make use of such information (let alone the user) in order to take the appropriate action.

Therefore the Mozart failure model is designed to fulfill the following

- Reflect failure to the language level
- Take no irrevocable action
- Provide both eager and lazy failure detection
- Provide the programmer with the ability to replace failed actions by other actions

The Mozart programmer deals with language entities. Failure is detected on this level as well. Language entities are either normal, permanently failed, or temporarily failed. The ultimate cause of such failures is, of course, either the crash of a site or some network problem. The latter condition may, but need not be, temporary. Network failures may mask crashed sites. However, the programmer need not think in terms of sites and networks, or even be aware of them. Permanent failed entities will never work properly, while temporary failed entities might recover.

The system takes no irrevocable action upon detecting failure. Threads that attempt to operate on failed entities merely suspend (or more precisely the system can be configured for this behavior). The are good reasons for wanting to allow this. The reason for this is very clear when dealing with temporary failures. If and when the network repairs itself the operation is transparently resumed. Appropriate time-outs are application-dependent, and indeed coarse-grained fault-tolerance might measure progress or lack thereof on a much higher level. When and if the current activity is to be aborted a group of such suspended threads will be terminated.

Eager fault detection is managed by a mechanism called watchers. The programmer attaches such watchers to entities - if the entity enters the failed state that the watcher is configured for, the watcher procedure will be invoked in its own thread. Lazy fault detection (called handlers) detects failed entities when operations on them are attempted. The attempted operation is replaced by the handler procedure. A common type of handler merely injects an exception into the calling thread, but the handler procedure might also replace the attempted operation by an alternative operation (e.g.

instead of invoking one service instance invoking another instance known to be equivalent). Another kind of handler is one that upon temporary failure sets an application-dependent timer (in a separate thread) that will abort the operation (i.e. inject an exception) if no progress is made within the programmed time.

Mozart provides the programmer with a model to reason about and deal with failure on the language level, i.e. without breaking network-transparency. Of course, creating fault-tolerant abstractions and applications is still very difficult. If the programmer is not successful in masking failure the user will still be confused. To paraphrase Lamport 'you know you are dealing with a distributed system when some distributed object that you have no idea what it is supposed to do tells you it's broken'.

2.4.3 Transparency Classification

We can now complete our classification of the traditional transparencies of distributed computing from the language point-of-view. The defintions of the transparencies were given in section 2.1.

We have three groups. The first group, listed below, are properties of the network-transparent language (requiring considerable support by the system).

access transparency

location transparency

concurrency transparency

mobility transparency

Mozart exhibits all these transparencies in full.

The second group relates to mechanisms that the distributed programming system may or may not use. The motivation to use them is to improve the non-functional properties of the system. There should be no side-effects, i.e. language network-transparency is not broken by the introduction of these mechanisms.

replication transparency

relocation transparency

migration transparency

Mozart uses the mechanisms freely. For instance, immutable are freely replicated, and object-state both migrates and gets relocated.

Finally, there are transparencies that may or may not be desired on the application level. Some may be common enough to be put in libraries. The distributed programming system should have the necessary support to be able to achieve these transparencies on the the application level.

failure transparency

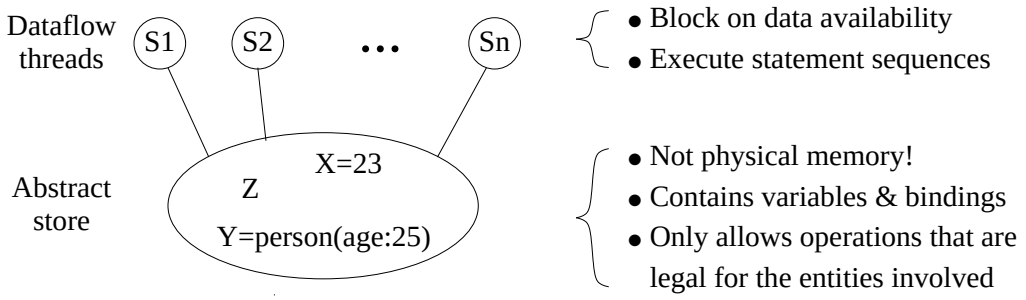


Figure 2.1: Computation model of OPM

persistence transparency

performance transparency

scaling transparency

If the Mozart system has the necessary support for programming fault-tolerant applications (i.e. achieving failure on the application level) is still somewhat of an open question (see also chapter 17). Other than that we see no major difficulty with the language constructs. Of course, there are improvements that can be made to the system that impact performance and scaling, ranging from JIT-compilation to reworking some of the protocols for better scaling.

2.5 The Oz Programming Language

Mozart is based on the concurrent programming language Oz. Here we give a brief overview of the language. A fuller description is given in paper 2 (chapter 7). For full details see www.mozart-oz.org.

Oz is a rich language built from a small set of powerful ideas. We summarize its programming model.

The roots of Oz are in concurrent and constraint logic programming. But Oz provides a firm foundation for *all* facets of computation, not just for a declarative subset. The semantics should be fully defined and bring the operational aspects out into the open. For example, concurrency and stateful execution make it easy to write programs that interact with the external world [58]. True higher-orderness results in compact, modular programs [4].

The basic computation model of Mozart is an abstract store observed by dataflow threads (see Figure 2.1). A thread executes a sequence of statements and blocks on the availability of data. The store is not physical memory. It only allows operations that are legal for the entities involved, i.e., no type casting or address calculation. The store has three compartments: the constraint store, containing variables and their bindings,

S ::=	$S\ S$	Sequence
|	X=f(l_1:Y_1 ... l_n:Y_n) |	Value
	X=<number> | X=<atom> | {NewName X}	
|	**local** X_1 ... X_n **in** S **end** | X=Y	Variable
|	**proc** {X Y_1 ... Y_n} S **end** | {X Y_1 ... Y_n}	Procedure
|	{NewCell Y X} | {Exchange X Y Z} | {Access X Y}	State
|	{NewPort Y X} | {Send X Y}	Ports
|	**case** X==Y **then** S **else** S **end**	Conditional
|	**thread** S **end** | {GetThreadId X}	Thread
|	**try** S **catch** X **then** S **end** | **raise** X **end**	Exception

Figure 2.2: Kernel language of OPM

the procedure store, containing procedure definitions, and the cell store, containing mutable pointers ("cells"). The constraint and procedure stores are monotonic, i.e., information can only be added to them, not changed or removed. Threads block on availability of data in the constraint store.

The threads execute a kernel language called Oz Programming Model (OPM) [116]. We briefly describe the OPM constructs as given in Figure 2.5. Statement sequences are reduced sequentially inside a thread. Values (records, numbers, etc.) are introduced explicitly and can be equated to variables. All variables are logic variables, declared in an explicit scope defined by the **local** construct. Procedures are defined at run-time with the **proc** construct and referred to by a variable. Procedure applications block until their first argument refers to a procedure. State is created explicitly by NewCell, which creates a *cell*, an updatable pointer into the constraint store. Cells are updated by Exchange and read by Access. Conditionals use the keyword **case** and block until the condition is true or false in the constraint store. Threads are created explicitly with the **thread** construct and have their own identifier. Exception handling is dynamically scoped and uses the **try** and **raise** constructs. Ports

A port is an asynchronous channel that supports many-to-one communication. A port P encapsulates a stream S. A stream is a list with unbound tail. The operation {Send P M} adds M to the end of S. Successive sends from the same thread appear in the order they were sent. By sharing the stream S between threads many-to-many communication is obtained.

Full Mozart/Oz is defined by transforming all its statements into this basic model. Full Oz supports idioms such as objects, classes, reentrant locks, and ports [116, 132]. The system implements them efficiently while respecting their definitions. We define the essence of these idioms as follows. For clarity, we have made small conceptual simplifications. Full definitions may be found in [51].

- **Object**. An object is essentially a one-argument procedure {Obj M} that references a cell, which is hidden by lexical scoping. The cell holds the object's state. The argument M indexes into the method table. A method is a procedure that is

given the message and the object state, and calculates the new state.

- **Class**. A class is essentially a record that contains the method table and attribute names. When a class is defined, multiple inheritance conflicts are resolved to build its method table. Unlike Java, classes in Oz are pure values, i.e., they are stateless.

- **Reentrant lock**. A reentrant lock is essentially a one-argument procedure `{Lck P}` used for explicit mutual exclusion, e.g., of method bodies in objects used concurrently. `P` is a zero-argument procedure defining the critical section. Reentrant means that the same thread is allowed to reenter the lock. Calls to the lock may therefore be nested. The lock is released automatically if the thread in the body terminates or raises an exception that escapes the lock body.

2.6 Language Extensions

We have shown and argued that a network-transparent and network-aware distributed programming system based on a good concurent programming model will take you very far. However, this does not mean that this is enough. Developing a programming system that 'presents itself to the programmer as if it were a single computer system' is not sufficient.

The reason for this is quite simple. There are facets of computing that are either not visible, not important, or not natural on a single computer system. These facets will, however, become an intrinsic part of distributed programming, and will be reflected in the distributed programming language.

It is, we think, unclear exactly what extensions will ultimately be needed on the programming system level (maybe they can be dealt with on top of the programmng system). We do not claim that Mozart is complete in this sense. See also section 17.

The failure monitoring constructs that were described in section 3.2.1 belong to this category. They were put into the language to deal with partial failure, which is not visible (or not possible) on a single computer system.

In the next subsection we describe another very important language extension.

2.6.1 Open Computing

One of the most important extensions to the programming language Oz to make it a distributed programming language was to make provisions for open computing. Open computing means that running applications must be able to establish connections with computations that have been started independently across the net.

This is an example of a need that is not natural on the single computer system where the user/programmer has total control.

Mozart uses a ticket-based mechanism to establish connections between independent sites. One site (called the server site) creates a ticket with which other sites (called client sites) can establish a connection. The ticket is a character string which can be

stored and transported through all media that can handle text, e.g., phone lines, electronic mail, paper, and so forth.

The ticket identifies both the server site and the language entity to which a remote reference will be made. Independent connections can be made to different entities on the same site. Establishing a connection has two effects. First, the sites connect by means of a network protocol (e.g., TCP). Second, in the Mozart computation space, a reference is created on the client site to a language entity on the server site. The second effect can be implemented by various means, i.e., by passing a zero-argument procedure, by unifying two variables, or by passing a port which is then used to send further values. Once an initial connection is established, then further connections as desired by applications can be built from the programming abstractions available in Oz. For example, it is possible to define a class `C` on one site, pass `C` to another site, define a class `D` inheriting from `C` on that site, and pass `D` back to the original site. This works because Mozart is fully network-transparent.

Mozart features two different types of tickets: one-shot tickets that are valid for establishing a single connection only (one-to-one connections), and many-shot tickets that allow multiple connections to the ticket's server (many-to-one connections). Tickets are used in the example program of paper 1(chapter 6).

Chapter 3

Distributed Programming Systems

A distributed programming system is a realization of a distributed programming language. In the previous chapter we presented the language aspects of the Mozart programming system. In this chapter we turn our attention to the system. In the course of the Mozart work we faced a number of severe challenges that had to be met. These challenges and how we overcame this is the subject of this chapter. This chapter is supported by papers 2-4 (chapters 7 to 9).

We worked with the following design and implementation principles:

1. Maximal network transparency
2. Good Network awareness
3. Open and dynamic computing
4. Efficient local execution
5. Minimal protocol infrastructure
6. Minimal latency for reference passing

The first principle was thoroughly discussed in the previous chapter. There we also introduced the principle of *network awareness* which is here used in the sense of paper 1 (chapter 6). Network awareness may be further subdivided into a true awareness aspect and a control aspect. The true awareness aspect is that the system behaves in a predictable manner, and that associated with the system are programming models over the non-functional aspects of the system, e.g. performance and failure. (By performance here we are thinking mainly of distribution aspects such as number of network hops, number and size of messages, and latency). The control aspect reflects that the programmer must be able to control where computations take place, be able to tune the program with respect to performance (according to expected or observed application-specific usage patterns), and be able to deal with failure on the language level.

The third principle has also already been discussed to some degree. Mozart caters for open and dynamic distributed applications. It is not restricted to LANs, but is

suitable for WANs and the Internet. The machines involved in a distributed application, at any point in time, is a dynamic property. The machines do not have to be known a priori, machines may continually join an application (e.g. via tickets), or leave (e.g. as shared references go out of scope). In one sense, as we shall see, Mozart also scales perfectly (see section 3.1.1).

The last three design principles may also be seen as aspects of network awareness, but their importance motivates listing them explicitly here. These three principles are, as we shall see, important for either performance and/or failure vulnerability.

3.1 Network transparency

3.1.1 The Protocol Infrastructure

Network transparency meant that we had to take into account that all language entities that in the concurrent centralized system could be shared between threads on the same machine could now be shared between threads on different machines. This meant that we had to, for many of the language entities, devise or find protocols (or distributed algorithms) that would coordinate operations on sites that reference the entity. Clearly, we wanted the most efficient (in terms of messages and network hops) without sacrificing semantics (consistency).

An important design principle in the work was not to rely on any kind of central authority or out-of-band service. This can be formulated precisely. Whatever the entity type if a language entity is shared between a set of sites then only those sites and no others will be involved in the protocol. Actually, for practical reasons, we relaxed this to, 'only those sites that reference the entity and the creation site will be involved the protocol'. (At the time it seemed that usually the site that creates an entity participates in the sharing throughout the entity lifetime).

Note that this differs from many other distributed programming systems. In Globe [122], for example, there is a central and separate naming and location service that has to be maintained separately. In this sense Mozart is self-organizing. Note that in running distributed applications the sites involved in the application may be many more than share any one particular entity. The sites are still linked and can influence one another indirectly via chains of shared entities.

Note that a Mozart application, in one sense, scales perfectly. A loosely-connected application (i.e. connected only by virtue of chains of shared entities) can potentially number in the millions of sites or machines. Mozart allows for this. Of course, it is problematic if a millions of sites share the same object or logical variable, as there are limits in the scalability of the protocol coordinating any one particular (non-stateless) language entity.

3.1.2 Development of Protocols

We developed a set of protocols to handle the different types of language entities. Some of these protocols were fairly simple and standard, others were challenging. An example of the first type is the protocol supporting ports (the asynchronous message-passing primitive). The mobile state protocol and the variable protocol exemplify the second type.

We developed an entirely new protocol for single-assignment or logical variables that realized distributed unification efficiently. This is described in paper 3 (chapter 8).

We developed a new protocol for dealing with Oz cells. Oz cells form the basis of mutable state (and objects) in Mozart. This is described in paper 2 (chapter 7).

The stateless language entities of Mozart occur in two varieties based on the supported equality relationship. Records, atoms, integer and floats have structural equality. Procedure, classes, and object-records have token equality, i.e. are only equal if they have the same source. Network transparency implies that token equality must be preserved, i.e. the system must recognize that equality of repeated imports of the same (via token equality) data structure. This was achieved by the use of global names, and had the additional benefit that irrespective of history there is at most one copy per site of the data structures corresponding to code (i.e. classes and procedures).

3.1.3 Marshaling

Stateless language entities (records, atoms, integers, floats, classes and procedures) and snapshot state (as part of the mobile state protocol) would need to be transferred between sites. We developed an efficient (i.e compact) marshaling format and implemented a marshaler/unmarshaler to convert data structures in memory into linearized form and vice versa. The marshaling format recognizes co-references for compact marshaling (this was also necessary as Oz allows for cyclic data structures).

One of the complications of marshaling is the risk of memory explosion. Sites and networks vary in capacity and marshaled data structures consume memory space. Protocol operation can quite easily deliver more messages (and in particular messages containing Mozart data structures) than the network can handle. The problem is twofold. First, the marshaled linearized representation of language entities tends to be much larger than the the space consumed by the entity in program memory. Second, the space in the marshaling buffer is space that is used in addition to that in memory. Effectively there are two copies of the same entity in different forms.

An important design decision was therefore to queue messages in unmarshaled form, i.e. to implement late marshaling. This is also described in the section on the messaging or network layer (section 3.2.3). Entities are thus not marshaled when the channel buffer (one for each other site that a site is connected to) exceeds some critical size.

The story does not really end there. Individual messages may contain very large data structures that take considerable space in marshaled form. For this reasons, the marshaling framework in Mozart was later extended (in joint work including the au-

thor) to provide for full incremental marshaling. This is beyond the scope of this thesis but is described in [101]. With incremental marshaling Mozart only needs a constant buffer space per channel.

Asynchronous operations (i.e. sending to a port) might potentially cause memory explosion - in program memory this time, not just as marshaled data. It was therefore necessary to provide feedback from the marshaler to execution, threads engaged in asynchronous operations are suspended or slowed when the number or size of undelivered messages grows too large.

3.1.4 Garbage Collection

Mozart/Oz is a high-level programming language and has automatic memory reclamation or garbage-collection. For complete network transparency memory reclamation also needs to work on entities that have been shared.

We developed a distributed garbage collection scheme based on weighted reference counting (called a distributed credit mechanism in paper 1). This collector is guaranteed to reclaim all acyclic garbage (modulo failure).

Weighted reference counting allows a site holding a reference to freely export this reference without coordinating with the home of the entity (or any type of central authority) for minimal latency. It merely divides the weight associated with the entity between itself and the exported reference. Of course, a problem occurs when the weight cannot be further subdivided.

Weighted reference counting was modified in the initial Mozart release to be able to handle arbitrary reference passing patterns, i.e. avoiding the problem of not being able to further divide the weight. The scheme suffered from complexity and was later superseded by a better algorithm (co-designed by the author) called fractional weighted reference counting published in [74].

Our distributed garbage collector does not so much recognize that shared entities become garbage, but rather detects the situation when there is (at most) one site referencing the entity. The entity can then be localized for efficient local execution (or possibly reclamation by the local garbage collector).

The weighted reference counting garbage collector breaks in the face of site failure. Weight is lost and garbage will then never be reclaimed. More sophisticated safe distributed garbage collection schemes to deal with this are both expensive and break one or other of our design principles [98]. For instance, they may demand that all sites know about each other, which violates openness and self-organization. Later as is suggested in paper 1 (chapter 6), a time-lease based garbage collection scheme was implemented as an alternative (programmer's choice). Note that, time-lease garbage collection may break network-transparency, as long delays may cause entities to be prematurely reclaimed.

3.2 Network Awareness

Here we are concerned with the non-functional properties of the system. We consider performance and failure issues. Here and in paper 1 (chapter 6) the term network awareness in used as term to denote both proper awareness issues and control issues. In part III we separate and make a clear division between awareness and control issues.

Network-awareness covers system aspects properties that make the system predictable in the terms of performance, failure, etc. Network-awareness also cover system properties that give the programmer the necessary control to adapt and optimize the application depending on deployment, expected usage patterns, etc. This control aspect of network-awareness requires that the system is reasonably efficient and fault-tolerant to begin with.

The protocols were designed with network-awareness in mind. This is discussed both in the next chapter (comparing Mozart with other systems) and in the papers (part II).

In the rest of this section we consider other aspects of network-awareness that we dealt with in the course of the Mozart work.

3.2.1 Failure Model

The failure model was described in the previous section. The system needed considerable instrumentation to realize this model.

The key to realization of the failure model was to map the underlying fault detection in the network layer on the level of sites and connections between sites onto failure on the level of language entities. This was done efficiently in that when failure was handled exclusively by handlers (i.e. lazily with no eager watchers installed) there was no impact on system performance in the absence of failure.

Eager failure detection, via watchers, had some minor impact on system performance. It requires the continuous monitoring of communication channels possibly above that which is required for protocol operation. This is also discussed in section 3.2.3.

The protocols needed to be instrumented to provide the information required for entity fault detection. In the case of the mobile state state protocol, considerable modifications needed to be made on the original protocol as described in paper 2 (chapter 7). The modifications are described in paper 4 (chapter 9).

Of course, failure detection on language entities is only as reliable as the underlying network layer can deliver reliable information as to the failure of sites and communication channels.

The handler and watcher mechanism were made primitive in the language and system, and integrated into the centralized execution engine. It was necessary to be able to install handlers and watchers on a local language entity as the entity might later become distributed. To allow handler/watcher installation only on distributed entities would violate the good concurrency properties of the base language (introducing possible race conditions). A handler/watcher installed on a local language entity that ends

up never being distributed has, of course, no effect.

3.2.2 Efficient Local Execution

A major design constraint was that operations on local language entities (i.e. only referenced from a single site) should be affected as little as possible. Ideally there would be no difference in performance in Mozart as compared to centralized Oz. In practice, we found that we pay cost on the order of 1%.

Note that the *localness* of a language entity is a dynamic property. During the course of execution local language entities may or may not become globalized, i.e shared. For example, a site that has references to a local object and globalized object may update the state (attribute) of the globalized object to reference the local object. The previously local object is now globalized and may be accessed by threads on other sites. Note that the fact that an entity is globalized does not mean that the entity is is not available locally, but only that operations must go through the coordination or protocol logic. For instance, in the mobile object protocol of papers 2 and 4, the object state may at invocation time be cached locally.

In practice, the design constraint that we worked with was that operations on entities that are at operation-time local should be virtually unchanged. This means that the performance of operations on an entity is independent of the entity's history. This requires the cooperation of the distributed garbage collector.

A Mozart site thus carefully maintains a dynamic distinction between local and globalized entities. Operations on local entities are done efficiently and without any synchronization between sites.

Two observations may be made here. First, we make good use of the language property of reference security here. The only way that a local entity can become globalized is by action at the site that holds the local entity. No uncoordinated action taken by any other site can achieve this as no other site holds a reference to the entity.

Second, the problems of distributed shared memory (DSM) are avoided. Distributed shared memory (a language-independent middleware) must also cater for imperative languages (e.g. concurrent C) that allow for pointer arithmetic. All memory is potentially addressable by all sites. This would be as if all entities were globalized. DSM systems also suffer from the problem of false sharing [122], which is also avoided - in Mozart that which seems to be shared is always truly shared.

3.2.3 Messaging and Network layer

Mozart uses TCP (as supported by Linux, Solaris and Windows OS) as the underlying communication mechanism between sites. Mozart threads are not OS-threads (this is what makes concurrency scalable in Mozart) but are part of the Mozart runtime. We could therefore not use blocking TCP. But this was by no means enough and above TCP we found it necessary to build a fairly complex messaging-layer. The messaging layer did the following:

Trsnslation of TCP Error Codes: The TCP error codes had to be translated into permanent and temporary failure of sites, which in turn, in other layers, would be mapped onto entity failures. This translation is conservative and a permanent failure is only declared when crash failure can be safely determined. Determining site failure could also be established in other ways. For instance, when one Mozart OS-process crashes and another is started with the same IP-address and port that the permanent failure of the crashed site will be determined when a connection is established with the newly started site.

Recovery: In Mozart, time-outs, in the sense of when the suspended operation on a temporary failed entity is aborted, is determined by the programmer. TCP can thus time-out before Mozart time-outs. This requires that the network layer can recover from broken TCP connections with all that this entails in terms of acknowledgments, buffering, and resending on the network layer level. (The network layer here actually duplicates on another level many of the fundamental TCP mechanisms).

Sharing of connection resources: Sites have logical connections to other sites. Depending on entity type and the consistency protocol associated with the entity, sites have a logical connection to some or all of the sites that share the same language entity. A site therefore potentially has very many logical connections. At any one time some logical connections are passive in that there is currently no need to communicate with the site on the other end. This is quite common as sites hold references to object and ports that are seldom used. Other logical connection are active in the sense that there currently is a need to communicate with that site (i.e. messages to that site are currently queued). Finally there are physical connections in the sense that TCP connections are established between sites.

Most operating systems limit the number of physical connections that may be maintained. Mozart therefore shares the limited physical connection slots among the active logical connections. The messaging layer therefore contains queues and its own scheduler.

Watcher instrumentation: Watchers (for eager fault detection) have an affect on connection-resource sharing. Their operation will be mapped to the need for regular monitoring of a number of logical connections. If these logical connections are not made active by normal execution involving shared language entities, then they must be made active anyway.

Queuing and Buffering: In general, messages may be queued waiting for scheduling. Some of the messages contain language entities (Oz terms). To save memory the language entities are not marshaled when referenced from the queue. The queue cooperates with the local garbage collector.

In the Mozart implementation of 2000 once a message (containing a reference to a language entity) was scheduled and marshaling was begun, marshaling was

complete without interruption. Some messages may be very large (e.g. sending a large tree data structure to a port), and this usually requires buffering in the Mozart runtime as the underlying TCP, with limited buffering, cannot hold the entire message. It is buffered in the Mozart system and the buffer may grow and shrink depending on current needs. TCP cannot, of course, free its buffer space before it receives acknowledgment from the receiver.

Time-sharing between communication and execution: In centralized Mozart/Oz the internal scheduler manages the time-sharing of runnable threads. In distributed Mozart there is the need for time-sharing between messaging layer operation (including marshaling) and thread execution.

3.2.4 Dealing with Localized Resources

The Mozart Programming System like other programming systems makes use of standard operation system resources. Typical resources are the file system and the window system (graphics). These *resources* can be seen as references from within the Mozart/Oz computations world to the outside world and the outside world is, of course, not network transparent. So the question is, what should be done when such references are exported from one site to another?

With some analysis it quickly becomes apparent that the desired scoping is very unclear. This is not the case with the normal language entities of Oz, where lexical scoping is the rule. Consider, for example, the case when code (e.g. class or procedure) that encapsulates a reference to the file system is exported over the Internet from one site to another. What is the programmer's intent? Is the intention to use the file system at the sending site or the receiving site? In other words, do we want static linking or dynamic linking? This is just one case where the traditional constructs of centralized programming systems underspecify requirements. The necessary distinctions are just not there.

A time-consuming partial solution to the resource problem would have been to rework the Mozart/Oz resource model to capture the distinction between static and dynamic resources. This we did not do. Indeed, an attempt to do so would suffer from many complications. Statically linked resources may have a wider scope than just the particular site or operating system process. For example, the same file system may be used directly by sites on the same LAN. Some resources are stateful (e.g. an open file) and cannot be made dynamic.

Instead, we ensured that the programmer was given the necessary means to realize static linking, where this was what was wanted, and dynamic linking, where this was what was wanted. Static linking is achieved by using the stationary object abstraction as described in papers 1 and 2 (chapters 6 and 7. For instance, a statically linked file system would transfer file operations back to the original site. Dynamic linking is achieved by using functors [28].

This left us with the question of what to do when the programmer does nothing and shares a resource reference directly without making the necessary provision for

static or dynamic linking. We took the safe course, and these resources simply do not work outside the original site (i.e. give rise to exceptions on attempted use). This has the added benefit that programmer oversight does not introduce a security hole, but causes the application to fail (or more precisely, the thread). Resources can be harmlessly exported (maybe the reference is never actually used). For reasons of maximally transparency, resources that are exported and then reimported to the original site do work.

Note that in Mozart, code (i.e. classes and procedures) are first-class values and have full distribution support for token equality. When these code constructs are sent across the net this does not mean that the actual code needs to be shipped (marshaled and unmarshaled). The receiving site may very well, and this is quite common, already have the required code, which is then linked in. This has the performance advantages of dynamic linking but is safe.

Dynamic linking of code, however, is very important for code revision. New and by token-equality different code components should be able to connect as replacements. This is also handled by functors were code can explicitly be equated by name. (For some of the problems of dealing with code in other systems see also section 16.4.)

Chapter 4

Related Work

In this chapter we relate Mozart to other distributed programming systems.

The first section covers systems where the centralized programming model and the model over distribution support are completely different. This kind of system can be easily constructed by making use of some what are called 'language-independent' distribution support tool (or middleware) together with some centralized programming system. The section is titled the 'two-headed beast', because it makes use of two programming models that are typically very different. This makes programming beastly - much like trying to control a two-headed beast where each head has its own idea of where to go.

The second section covers distributed programming systems that to different degrees avoid the problem with the 'two-headed' approach by incorporating centralized programming and distributed programming in the same programming language. In part III this approach is called the integrated approach (see section 15.3).

4.1 The Two-Headed Beast

The most straightforward approach to distributed programming is to use separate models, tools, and systems for centralized programming on the one hand and distributed programming on the other. There are, however, many disadvantages to this approach:

Multiplicity of models: The programmer must work with two very different programming models and be extremely careful when working in the intersection of the two.

Difficult to change distribution structure: Any changes in the distribution structure of the application, i.e. how threads are distributed between sites, will require major reworking of the application.

Different data formats: The data formats in the centralized programming model and the distribution model are different and any exchange of information between the two needs to be converted. This is less serious in that this may be dealt with on the library level.

Mismatch of concepts: This last factor depends on the nature of the distribution support tool. The centralized and distributed programming model may be so different so as to violate each other's basic assumptions. This will become clearer when we consider individual systems.

We call this approach the *two-headed approach*.

4.1.1 Message-passing systems

Using a message-passing system [122, 29] together with a centralized programming language/system falls into the two-headed approach (except for Erlang as will be described at the end of this subsection).

The simplest message-passing system is a centralized programming language together with the socket library (i.e. TCP). More sophisticated message-passing support can be found in MPI or MOM (message-oriented middleware) [122]. Support for data conversion between different formats can be included [104, 2]. However, the disadvantages of multiplicity of models, and difficulty in changing the distribution structure remain.

Usually there is a serious mismatch of concepts between the message-passing middleware and the centralized programming system. This is the case when the centralized programming system is based on mutable state programming. This includes most programming systems, and all the really popular ones (some exceptions are functional programming languages, the declarative subset of Mozart, and Erlang). Messages are stateless (immutable) and once sent cannot be unsent. The application developed will have to work with local mutable state on the one hand and immutable messages and undoable message-sending (i.e. you cannot retract sent messages) on the other. This is a serious mismatch.

What to do about this mismatch was actually investigated in the context of development of Oz (both before Mozart and early on during Mozart work when some smaller semantic changes in the language were made). You do not need to go to distribution - the mismatch can be investigated in a concurrent programming system that caters for both mutable state and declarative programming (including message-sending). The conclusions were:

Declarative concurrency: Declarative concurrency, which includes message-sending but not mutable state, is all that is needed for many concurrent applications. The advantages of this style are many, but most important may be that it lessens the risk of deadlock and race conditions. A full description can be found in [129]

Encapsulated state: There is a mismatch between the declarative subset and mutable state, but this is almost non-existent when all mutable state is kept local to a single thread.

Shared state: There is a fundamental mismatch but sometimes one needs shared state. There are techniques to minimize the dangers of the mismatch but programmers need to be careful.

One of our major criticisms of so many tools and programming system extensions for distributed programming is that they are presented as simple add-ons and libraries but actually introduce significant complexities. These complexities will bite the programmer as soon as he tries to go much beyond the simplest of applications. In Mozart these complexities are in the language, put up front, and made visible. The language can be mastered and then, by virtue of network-transparency, distribution is obtained almost for free.

At this point we need to discuss Erlang [138]. Erlang is a special case as the basic centralized (concurrent) programming model is based on message-passing. Adding distributed message-passing to concurrent Erlang does not, of course, add any new concepts. Distributed Erlang is thus not *two-headed* and does not suffer from its disadvantages. Erlang, like Mozart, has a declarative concurrent base. However, in Erlang there is no provision for shared state at all, and the only type of encapsulated state is on the individual process level (process dictionaries). Thus shared state is forbidden[1].

4.1.2 Database Approach

Another mixed approach is to use some type of shared data repository. Threads and sites coordinate by inserting, updating, reading and removing data from this repository.

This approach has many of the same disadvantages of the message-passing approach, such as multiplicity of models, and often, mismatch of programming concepts. For true database applications where the application basically does nothing except access and update the data repository the mismatch critique does not apply. This is not the type of application that we targeted with the Mozart work. In database applications the complexity is on the database side. Consistency, fault-tolerance and scalability issues are all important. Databases have, of course, been extensively researched.

Interesting in this regard is Linda [23] which provides operations to insert and extract immutable data items, called “tuples,” from a shared space. This is called a coordination model, and it can be added to any existing language. The fact that the data is immutable makes achieving consistency easier, and improves performance considerably compared to traditional updateable databases. One is reminded of Mozart where the language distinction between immutable, mutable, and single-assignment is leveraged for good performance. Note that Linda also suffers from the multiplicity of models. Once again, in the context of programming languages centered around manipulation of mutable state there is also a mismatch of concepts.

Linda can also be seen as form or library-based distributed-shared memory [122]. Unlike page-based (low-level) distributed shared memory only certain abstractions may be shared and given to DSM middleware to maintain consistency. Another example is Orca [14] where network pointers, called “general graphs,” and shared objects are provided. Orca is designed for parallel applications (requires LAN support to work well). Library-based DSM also suffers from the multiplicity of models.

In a sense Mozart also follows the library-based DSM approach. In this view the

[1] Many Erlang applcations do however make use of an external database

shared computation space of Mozart is a DSM layer that is designed to support **all** language entities. There is no distinction between entities that may be shared and those that may be not. This is vital for network-transparency.

Also, the Mozart DSM layer leverages the invariants associated with language entity types to provide the best consistency protocol. Stateless language entities can be freely replicated between sites. Single-assignment (data-flow) variables are intermediate in complexity between immutable and mutable data structures. Once this has been done the Mozart DSM is finished with the entity and the entity is indistinguishable from any other local entity.

From this point of view the Mozart DSM layer is extended to provide functionality that is not part of traditional DSMs. First, it supports single-assignment data in a strong form (logic variables) as well as other sharing protocols such as read-only data (values) and migratory data (objects). Second, the system is open: sites can connect and disconnect dynamically. Although not impossible, we do not know of any DSM system that possesses this property [2] . Third, the system is portable across a wide range of operating systems and processors. Fourth, the system supports precise failure detection.

4.2 The Language Approach

Here we compare with distributed programming systems that at least to some degree avoid the problem of the multiplicity of models as described in the previous section. We can compare systems and the associated languages based on:

- Language Expressivity: The most natural and common way of comparing the relative merits and demerits of programming languages.
- Network transparency: The extent to which the systems offer network transparency.
- Network awareness: Here we consider various non-functional properties of the system, such a predictability, performance (for distributed operations), and failure.

4.2.1 Expressivity

The expressivity that Mozart offers is mainly due to the fact that it is based on the concurrent programming language Oz. The expressivity carries over to distribution and because of network-transparency the programming model is not complicated by having to take into account non-transparencies.

Oz has a solid formal foundation that does not sacrifice expressiveness or efficient implementation. Oz is based on a higher-order, state-aware, concurrent constraint

[2] It is beyond the scope of this thesis but later work at SICS, with the involvement of the author has developed, for the first time, something along these lines. See [1]

computation model. Oz appears to the programmer as a concurrent object-oriented language that is every bit as advanced as modern languages such as Java. The current emulator-based implementation is as good or better than Java emulators [59, 58]. Standard techniques for concurrent object-oriented design apply to Oz [79]. Furthermore, Oz introduces powerful new techniques that are not supported by Java [51]. Higher-order means that the same mechanisms for sharing data between threads (and in the distributed setting between sites) can be used for sharing code.

Oz provides both synchronous (e.g. method invocation on objects) and asynchronous operations (via the message-sending construct ports). There is a noticeable difference in behavior between the two alternatives already in the centralized concurrent setting, but the need for both forms increases when distribution is added to the mix.

Oz is a state-aware and dataflow language. State-awareness means the language distinguishes between stateless data (e.g., procedures or values) and stateful data (e.g., objects). Dataflow synchronization allows to decouple calculating a value from passing it as reference. Data-flow synchronization between threads is also useful in that it lessens the risk for deadlock and race conditions that otherwise plague concurrent programming systems [129].

Oz is referentially secure and provides language security. References to all language entities are created and passed explicitly. An application cannot forge references nor access references that have not been explicitly given to it. The underlying representation of language entities is inaccessible to the programmer. Oz has an abstract store with lexical scoping and first-class procedures. These are essential properties to implement a capability-based security policy within the language [125, 134].

Object-oriented systems like Java-RMI [120], Emerald [71, 72], and Globe [122] do not provide higher-order programming nor data-flow variables and thus many of the programming techniques of Mozart are just not available. Neither do most object-oriented systems provide for asynchronous thread interaction. In the Java-based system Object Voyager [3] extensions were added to permit asynchronous method invocation, which if the method return void can be seen as message-passing. (Object Voyager is an example of an add-on, for distribution, that though it has major programming implications, is presented as if it were a very minor addition).

Obliq [21] shares with Mozart the notions of dynamic typing, concurrency, state awareness, and higher-orderness with distributed lexical scoping. There are however no provision for data-flow. Obliq like Emerald is also only object-based (i.e. not object-oriented in the sense of Mozart or Java).

Erlang [138] does not provide distributed objects nor data-flow variables. Thread interaction is limited to message-passing.

4.2.2 Transparency

Most distributed programming systems are only partially network-transparent. At first sight this is strange, given the consensus on the merits of transparency. The reason for this, in the author's opinion is threefold. First, few have taken the language view and

transparency is sought on the application level, as discussed in section 2.2.3. Second, network transparency is difficult to realize without sacrificing network awareness (e.g. reasonable performance). A network transparent solution with poor performance is of little interest. Third, many programming languages have limitations that just do not allow both good network transparency and good network awareness.

Partial network transparency means that changes in the distribution structure of the program often requires rewriting the application. In Java RMI [120, 90], for example, invoking a remote object where all method parameters are simple types is network-transparent, modulo failure and timing, just like Mozart. In this case Java RMI is maximally transparent. However when the parameters are (stateful) local objects the remote method invocation is not transparent, i.e. does not exhibit the same behavior as a local method invocation would. Any change in the distribution structure which places interacting threads that previously resided on the same machine on different machines runs a high risk of changing program behavior. There are many other non-transparencies in Java RMI. There are issues related to thread identity when a thread performs a remote invocation. In RMI re-entrant locking does not work when the call graph extends over the net (so changing an application's distribution structure can cause deadlock!).

Recently, there has been considerable research effort in java-based system to fix the transparency problems. The thread identity problem is fixed both in cJVM ([10]) and in [137]. Transparency is most complete in cJVM (which is designed exclusively for clusters). (Much of this work was done after the Mozart work and release).

In most distributed object systems there is a distinction beween objects that may be distributed and those that may not. This applies to Java, Obliq, Object Voyager and many others.

Distributed Erlang and Emerald are network-transparent, but the languages are less expressive than Mozart.

4.2.3 Network awareness

Here we consider both aspects of network awareness. First, we consider predictability. Does the system provide a model for the programmer to predict performance properties (i.e. network hops), and failure resilience? Does the system provide the programmer with the necessary control to achieve good performance.

Imperative Programming Languages and DSM

Languages that are not referentially secure (i.e. imperative with pointer arithmetic) suffer from the fact that the distinction between local and globalized (shared) entities is difficult to maintain. This has considerable impact on performance as described in section 3.2.2. Distributed programming systems based on imperative programming languages require distributed shared memory support.

Distributed shared memory (DSM) [29, 122] does provide a substrate for distributed programming. Page-based DSM does not provide predictable network awareness

and may perform badly (when part of the page is actually only used on one site). The units of distribution ("pages") do not correspond directly to language entities. This leads to false sharing. Munin [24], while page-based, provides programmer annotations for network awareness. A data item in memory can be annotated as read-only, migratory, write-shared, and so forth.

Distributed Object Systems

Many distributed object systems (e.g. JavaRMI [90, 120], Object Voyager [3]) only provide for stationary objects. In section 16.6.3 we analyze use cases for distributed object systems and see that for good performance under all usage patterns this is not enough. The migratory object protocol of Mozart (paper 2 in chapter 7), which in a sense caches the state of the object at the site invoking it, performs best under certain usage patterns.

Systems like Emerald and Obliq provide for objects that may be explicitly, by the programmer, moved from one site to another.

Java-based

Traditional Java RMI only provides for stationary objects which is inefficient for a wide range of applications (see section 16.6.3). The Java programming language (and the way it is typically used) makes it difficult to provide good transparency and good performance at the same time. This is because the language is not state-aware, there is no clear distinction between stateless and stateful language entities.

Java is most transparent when all the parameters in a remote invocation are stateless. This is the case when they are all simple types (e.g. numbers). This is also the case when they are stateless objects. There is only one way to express this in the language and that is to make use of objects where all instance variables are final. This is limited (there is no way to express that an object becomes stateless subsequent to object initialization). Also, in centralized programming, there are no semantic consequences of omitting the final declaration so programmers are only motivated to make this annotation for defensive programming. The practical consequence of this is that Java programs abound with stateless data structures masquerading as stateful ones.

If Java-based systems were to implement complete transparency then all objects except for those where all instance variables are final would upon sharing be made into shared objects (e.g. remote). All accesses except at the original site would require 2 network hops. Clearly this is unacceptable, performance-wise, if the object is stateless and only masquerading as a stateful object.

Practical use of Java-RMI is based on the programming discipline that stateful objects that will be shared should be declared (i.e. compiled) as remote objects, and those that are stateless when shared should not be declared as remote. The latter masquerading objects will when shared (e.g. used as a parameter in a remote method invocation) be copied, which, of course, is exactly the right thing to do. However, this is very risky and defensive programming is not possible.

In cJVM [10] where Java is made network-transparent they were forced to extend the language to get good network-awareness. They introduce extra annotations to provide the system with the information as to the statefulness of Java objects. Annotations may also be seen in [31]. It is unclear how cJVM could be extended to Internet applications.

In the interest of performance in the sense of network hops and network delays many other Java systems also extend the language. They may introduce asynchronous method invocation and futures [38].

In recent years (subsequent to Mozart) there are many attempts to improve Java. An interesting system is Java Party ([57],[97]). Distributed objects can be moved, and replicated safely under a read/write invalidation protocol. In recent years (beyond the scope of this thesis) Mozart has also been extended with read/write invalidation and explicitly movable distributed objects. In section 16.6.3 we identify four different protocols for distributed objects (state) that for good performance (hence, network awareness) distributed programming systems should offer.

Emerald

Emerald [72] is a statically typed concurrent object-based language that provides fine-grained object mobility [71, 72]. Emerald has distributed lexical scoping and is implemented efficiently with a two-level addressing scheme. Emerald is not an open system. Objects can be mutable or immutable, which is good from the point-of-view of network awareness.

Objects are stationary by default and explicit primitive operations exist to move them. Having an object reference gives both the right to call and to move the object; these rights are separated in Mozart. Immutable objects are copied when moved. Apart from object mobility, Emerald does not provide any special support for latency tolerance. There is no syntactic support for using objects as caches.

Moving a mutable object in Emerald is an atomic operation that clones the object on the destination site and aliases the original object to it. The result is that messages to the original object are passed to the new object through an aliasing indirection. If the object is migrated again, there will be two indirections, and so forth. The result is an aliasing chain depending on the object history. This makes for poor network awareness in the sense that number of hops is unpredictable and only bounded by the number of participating sites, which in open system is not limited.

The forwarding chain is lazily shortened in two ways. First, if the object returns to a previously visited site, then the chain is short-circuited. Second, all message replies inform the message sender of the object's new site. If the object is lost because a site failure induces a break in the aliasing chain, then a broadcast is used to find the object again. Using broadcast does not scale up to many sites, nor is it practical for Internet applications. As in Mozart, failure is detected for single objects.

Because of the aliasing chain and possible broadcasting, it is difficult or impossible to predict the network behavior in Emerald or to guarantee that an object is independent of third-party sites. These problems are solved in Mozart by using a manager node that

is known to all proxies (see paper 2 in chapter 7). This gives an upper bound of three on the number of network hops to get the object and guarantees that all third-party dependencies except for the manager site eventually disappear. Furthermore, the lack of an aliasing chain means that losing an object is so infrequent that it is considered as an object failure. There is therefore no need for a broadcast algorithm.

The Emerald system implements a distributed mark-and-sweep garbage collector. This algorithm is able to collect cross-site cycles, but it is significantly more complex than the Mozart mechanisms. It requires global synchronization and is thus not practical.

Obliq

Obliq has taken some steps toward the goal of conservatively extending language entities to a distributed setting. Obliq distinguishes between *values* and *locations*. Moving values causes them to be copied (replicated) between sites. Moving locations causes network references to them to be created.

Obliq objects are stationary. Object migration in Obliq can be implemented in two phases by cloning the object on another site and by aliasing the original object to the clone. These two phases must be executed atomically to preserve the integrity of the object's state. According to Obliq semantics, the object must therefore be declared as *serialized*. To be able to migrate these objects, the migration procedure must be executed internally by the object itself (be *self-inflicted*, in Obliq terminology). The result is an aliasing chain much like Emerald.

Erlang

Erlang is network-transparent but the language has its limitations. The only interactions between threads, and hence sites, is via message-sending. Thus the only language entities that are shared between sites are stateless messages and process references. There is no sharing of state and hence no caching of state as in the mobile object protocol of Mozart. For many applications this limitation carries a serious performance penalty as described in papers 1 and 2 (chapter 6 and 7). State may be programmed by letting a process hold the value of the state where accesses and updates are achieved by sending messages back and forth. This invariably rise to the same network communication pattern as stationary objects in Mozart or Java RMI.

Furthermore the message-passing primitive of Erlang offers only one-to-one and many-to-one communication. In Mozart one-to-many and many-to-many communication is achieved by the use of streams (in conjunction with ports for many-to-many communication). Communication with many readers must be programmed in Erlang and it is difficult or impossible to always achieve the wanted network communication patterns.

Erlang was not designed to be an open system, and is not higher-order which complicates dynamic sharing of code. (Higher-orderness or the ability to share code by the same mechanisms as whereby data is shared is very important for open distributed

systems but much less so for closed systems).

Data-flow variable

One of the interesting results from the Mozart work are the additional advantages of single-assignment for distribution above and beyond that of concurrent programming (all of which, also carry over to distribution). We have discussed their use for latency tolerance, but there is one more advantage.

Data-flow or single assignment variables can from the distribution point-of-view be seen as constrained state. The state is constrained in the way in which it can be updated or changed. The implementation of the associated protocol (chapter 8) leverages these constraints to make for a lightweight consistency protocol. True state can be updated many times and this requires a (relatively) heavyweight consistency protocol. This is not the case with single assignment variables. The protocol described in paper 3 can be compared to the closest related true state consistency protocol - invalidation with eager replication. Compared to true invalidation protocols the variable protocol leverages the single-assignment constraint and optimizes the eager invalidation protocol as follows:

- No invalidation messages need be sent
- The protocol infrastructure (proxy/manager network) can be dismantled automatically and safely after the one and only one eager propagation of the value.

The first point is due to the fact that all readers automatically synchronize (wait) when the variable is still unbound. The second point reflects the fact that the protocol infrastructure (i.e. the network of the manager and proxies) is only needed once.

4.3 Summary

We have compared Mozart with other distributed programming systems. We showed that the disadvantages of the two-headed approach to distributed programming are large. This is recognized not only by us, but by many others, hence the increasing interest in distributed programming system where the language is integrated with distribution support.

We have compared Mozart to other distributed programming systems based on these principles. We have analyzed this based on three criteria: language expressivity, network transparency, and network awareness. We have seen that Mozart is the most expressive, one of the most network transparent (particularly for Internet applications) and is among the best with respect to network awareness.

Chapter 5

Contributions by the Author

The thesis represents joint work done between 1995 and 2000.

The culmination of this work was the open-source release of the Mozart Programming System in 1999 and 2000 at www.mozart-oz.org.

The author was the main architect behind distribution in Mozart. The distribution support component of Mozart was mainly designed by the author. The author was the main implementor of this component in the earliest version. Later, the implementation team grew.

We first review some of the exceptions to the rule that the author was the main architect:

Oz: The centralized concurrent Oz language/system was designed and implemented by Prof. Gert Smolka's group in the early 90's. The core semantics of Oz was almost unchanged when Oz was extended for distribution to become Mozart.

The Marshaler: The marshaler/unmarshaler was designed and implemented by Ralf Scheidhauer (DFKI). The interface between the marshaler and the rest of the system was done by the author, including feedback flow control on asynchronous operations.

Token Equality: The naming mechanism to ensure token equality and preserve the at-most-one-copy property were developed by the author and Ralf Scheidhauer jointly

Protocols: The first version of the mobile state protocol (chapter 7) was developed by the author, but this was later improved jointly (mainly by Peter van Roy, Seif Haridi, and the author). The variable protocol of paper 3 (chapter 8) was developed jointly. The improved failure-detecting mobile state protocol (chapter 9) was developed mainly by the author. The other protocols mentioned in paper 1 (chapter 6) were developed mainly by the author.

Some particular contributions made by the author were:

Failure model: The author designed the failure model, and designed and implemented all the entity protocols to correctly detect entity failure.

Resource model: The author was also responsible for the resource model as described in the previous section.

Protocols: The lazy, eager and immediate protocols for stateless entities were developed by the author.

Garbage Collector: The author designed and implemented the distributed garbage collector.

Messaging and Network Layer: The author designed and implemented the network layer as described in the previous section.

Part II

The Papers

Overview of part II

The first paper was published 1998 in the journal of New Generation Computing. The paper presents Mozart (or Distributed Oz as it was called earlier) as a programming language designed specifically for distributed applications. Included is a short introduction to the concurrent programming language Oz. Thereafter the distributed programming language, Mozart, is introduced, and we see how the extensions for distribution in the language change virtually nothing (in the programming model). The only changes are in the programmer's model over certain non-functional characteristics (performance and failure model), which means that the distribution support is well-integrated into the concurrent programming model. A number of programming examples are included, demonstrating the ease of distributed programming in Mozart.

The remaining papers focus on the algorithms that underlie the implementation. The second paper, published in 1997 in TOPLAS, deals with distributed objects, or shared mutable language entities. In particular, the mobile state protocol (distributed algorithm) is presented. As is explained in the paper, the mobile state protocol is, for certain usage patters, optimal for shared mutables (e.g. distributed objects). Also shown in the paper is how to achieve one of the alternatives, a stationary state protocol, which is also optimal under different patterns of use.

The third paper, published 1999 in TOPLAS, is focused on the protocol underlying the sharing of data-flow variables. This, was the first effective algorithm for dealing with distributed unification. Interestingly, distribution actually increases the usefulness of data-flow variables over and beyond those that are found in centralized concurrent programming, where they simplify thread synchronization and enable concurrent declarative programming. Firstly, they are useful for latency-hiding, and secondly, the distribution protocols for single-assignment state (i.e. logical variables) are more efficient than the protocols for full-fledged (multiple-assignment) state.

The fourth paper (unpublished) presents a fault-tolerant (and fault-aware) version of the mobile state protocol. The version in the second paper is not practical in that certain failure conditions are not detectable (so the system cannot distinguish between latency and failure). The results described in this unpublished paper have been presented and published in slightly less detail in the paper, 'Lightweight Reliable Object Migration Protocol' [128]. This published paper is not included as it is an abbreviated version of what is covered by the second and fourth included paper.

Note that in some of the papers the Mozart was called Distributed Oz. This was before the public release of the Mozart system in 1999 and 2000.

Chapter 6

Programming Languages for Distributed Applications

Programming Languages for Distributed Applications

Seif Haridi[1], Peter Van Roy[2], Per Brand[3], and Christian Schulte[4]

[1] seif@sics.se, Swedish Institute of Computer Science, S-164 28 Kista, Sweden
[2] pvr@info.ucl.ac.be, Dép. INGI, Université catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium
[3] perbrand@sics.se, Swedish Institute of Computer Science, S-164 28 Kista, Sweden
[4] schulte@dfki.de, German Research Center for Artifi cial Intelligence (DFKI), D-66123 Saarbrücken, Germany

6.1 Abstract

Much progress has been made in distributed computing in the areas of distribution structure, open computing, fault tolerance, and security. Yet, writing distributed applications remains difficult because the programmer has to manage models of these areas explicitly. A major challenge is to integrate the four models into a coherent development platform. Such a platform should make it possible to cleanly separate an application's functionality from the other four concerns. Concurrent constraint programming, an evolution of concurrent logic programming, has both the expressiveness and the formal foundation needed to attempt this integration. As a first step, we have designed and built a platform that separates an application's functionality from its distribution structure. We have prototyped several collaborative tools with this platform, including a shared graphic editor whose design is presented in detail. The platform efficiently implements Distributed Oz, which extends the Oz language with constructs to express the distribution structure and with basic primitives for open computing, failure detection and handling, and resource control. Oz appears to the programmer as a concurrent object-oriented language with dataflow synchronization. Oz is based on a higher-order, state-aware, concurrent constraint computation model.

6.2 Introduction

Our society is becoming densely interconnected through computer networks. Transferring information around the world has become trivial. The Internet, built on top of the TCP/IP protocol family, has doubled in number of hosts every year since 1981, giving more than 20 million in 1997. Applications taking advantage of this new global organization are mushrooming. Collaborative work, from its humble beginnings as electronic mail and network newsgroups, is moving into workflow, multimedia, and true distributed environments [73, 41, 18, 13]. Heterogeneous and physically-separated information sources are being linked together. Tasks are being delegated across the network by means of agents [75]. Electronic commerce is possible through secure protocols.

Yet, despite this explosive development, distributed computing itself remains a major challenge. Why is this? A distributed system is a set of autonomous processes, linked together by a network [124, 85, 25]. To emphasize that these processes are not necessarily on the same machine, we call them *sites*. Such a system is fundamentally different from a single process. The system is inherently concurrent and nondeterministic. There is no global information nor global time. Communication delays between processes are unpredictable. There is a large probability of localized faults. The system is shared, so users must be protected from other users and their computational agents.

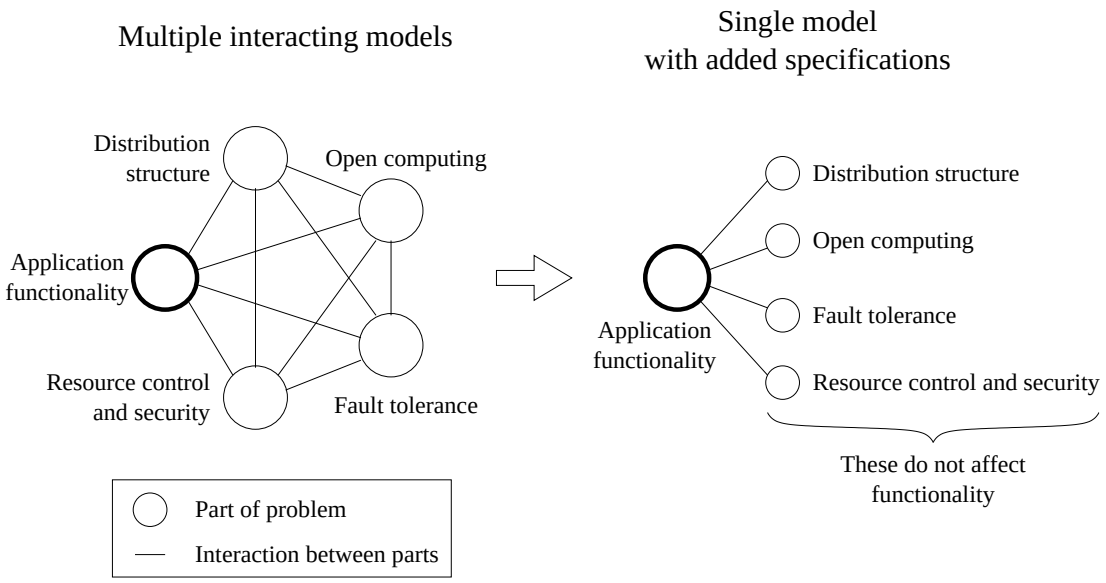


Figure 6.1: The challenge: simplifying distributed programming

6.2.1 Identifying the issues

A distributed application should have good perceived behavior, despite the vicissitudes of the underlying system. The application should have good performance, be dependable, and be easily interfaceable with other applications. How can we achieve this?

In the current state of the art, developing a distributed application with these properties requires specialist knowledge beyond that needed to develop an application on a single machine. For example, a new client-server application can be written with Java RMI [120, 90]. An existing application can be connected with another through a CORBA implementation (e.g., Orbix) [96]. Yet in both cases the tools are unsatisfactory. Simply reorganizing the distribution structure requires rewriting the application. Because the Java specification does not require time-sliced threads [46], doing such a reorganization in Java may require profound changes to the application. Furthermore, with each new problem that is addressed, e.g., adding a degree of fault tolerance, the complexity of the application increases. To master each new problem, the developer must learn a complex new tool in addition to the environment he or she already knows. A developer experienced only in centralized systems is not prepared.

Some progress has been made in integrating solutions to different problem areas into a single platform. For example, the Ericsson Open Telecom Platform (OTP) [36], based on the Erlang language [11, 138], integrates solutions for both distribution structure and fault tolerance. Erlang is network-transparent at the process level, i.e., messages between processes (a form of active objects) are sent in the same way independently of whether the processes are on the same or different sites. The OTP goes far beyond popular platforms such as Java [120, 90] and is being successfully used in

commercial telephony products, where reliability is paramount.

The success of the Erlang approach suggests applying it to the other problem areas of distributed computing. We identify four areas, namely distribution structure, open computing, fault tolerance, and security. If the application functionality is included, this means that the application designer has five concerns:

- **Functionality**: what the application does if all effects of distribution are disregarded.
- **Distribution structure**: the partitioning of the application over a set of sites.
- **Open computing**: the ability for independently-written applications to interact with each other in interesting ways.
- **Fault tolerance**: the ability for the application to continue providing its service despite partial failures.
- **Security**: the ability for the application to continue providing its service despite intentional interference. An important part of fault tolerance and security is **resource control**.

A possible approach is to separate the functionality from the other four concerns (see Figure 6.1). That is, we would like the bulk of an application's code to implement its functionality. Models of the four other concerns should be small and orthogonal additions. Can this approach work? This is a hard question and we do not yet have a complete answer. But some things can be said.

The first step is to separate the functionality from the distribution structure. We say that the system should be both network-transparent and network-aware. A system is *network-transparent* if computations behave in the same way independent of the distribution structure. Applications can be almost entirely programmed without considering the network. A system is *network-aware* if the programmer maintains full control over localization of computations and network communication patterns. The programmer decides where a computation is performed and controls the mobility and replication of data and code. This allows to obtain high performance.

6.2.2 Towards a solution

We have designed and implemented a language that successfully implements the first step, i.e., it completely separates the functionality from the distribution structure. The resulting language, Distributed Oz, is a conservative extension to the existing centralized Oz language [33]. Porting existing Oz programs to Distributed Oz requires essentially no effort. Why is Oz a good foundation for distributed programming? Because of three properties [118]:

- Oz has a solid formal foundation that does not sacrifice expressiveness or efficient implementation. Oz is based on a higher-order, state-aware, concurrent

constraint computation model. Oz appears to the programmer as a concurrent object-oriented language that is every bit as advanced as modern languages such as Java (see Section 6.4). The current emulator-based implementation is as good or better than Java emulators [59, 58]. Standard techniques for concurrent object-oriented design apply to Oz [79]. Furthermore, Oz introduces powerful new techniques that are not supported by Java [51].

- Oz is a state-aware and dataflow language. This helps give the programmer control over network communication patterns in a natural manner (see Section 6.5). State-awareness means the language distinguishes between stateless data (e.g., procedures or values), which can safely be copied to many sites, and stateful data (e.g., objects), which at any instant must reside on just one site [132]. Dataflow synchronization allows to decouple calculating a value from sending it across the network [53]. This is important for latency tolerance.

- Oz provides language security. That is, references to all language entities are created and passed explicitly. An application cannot forge references nor access references that have not been explicitly given to it. The underlying representation of language entities is inaccessible to the programmer. Oz has an abstract store with lexical scoping and first-class procedures (see Section 6.8). These are essential properties to implement a capability-based security policy within the language [125, 134].

Allowing a successful separation of functionality from distribution structure puts severe restrictions on a language. It would be almost impossible in C++ because the semantics are informal and unnecessarily complex and because the programmer has full access to all underlying representations [119]. It is possible in Oz because of the above three properties. So far, it has not been necessary to update the language semantics more than slightly to accommodate distribution.[5] This may change in the future. Furthermore, work is in progress to separate the functionality from the other three concerns. Currently, Distributed Oz provides the language semantics of Oz and complements it in four ways:

- It has constructs to express the distribution structure independently of the functionality (see Section 6.5). The shared graphic editor of Section 6.3 is designed according to this approach.

- It has primitives for open computing, based on the concept of *tickets* (see Section 6.6). This allows independently-running applications to connect and seamlessly exchange data and code.

- It has primitives for orthogonal failure detection and handling, based on the concepts of *handlers* and *watchers* (see Section 6.7). This allows to build a first level of fault tolerance.

[5] For example, ports have been changed to model asynchronous communication between sites [132].

- It supports a capability-based security policy and has primitives for resource control based on the concept of *virtual site* (see Section 6.8).

In Distributed Oz, developing an application is separated into two independent parts. First, only the logical architecture of the task is considered. The application is written in Oz without explicitly partitioning the computation among sites. One can check the *safety* and *liveness* properties[6] of the application by running it on one site. Second, the application is made *efficient* by specifying the network behavior of its entities. In particular, the mobility of stateful entities (objects) must be specified. For example, some objects may be placed on certain sites, and other objects may be given a particular mobile behavior (such as state caching).

The Distributed Oz implementation extends the Oz implementation with four non-trivial distributed algorithms. Three are designed for specific language entities, namely logic variables, object-records, and object-state. Logic variables are bound with a *variable binding* protocol (see Section 6.5.2). Object-records are duplicated among sites with a *lazy replication* protocol (see Section 6.5.3). Object-state moves between sites with a *mobile state* protocol (see Section 6.5.4). The fourth protocol is a distributed garbage collection algorithm using a credit mechanism (see Section 6.5.5). Garbage collection is part of the management of shared entities, and it therefore underlies the other three protocols.

6.2.3 Outline of the article

The rest of this article consists of six parts. Section 6.3 gives the design of a shared graphic editor in Distributed Oz. It shows how the separation between functionality and distribution works in practice. Section 6.4 gives an overview of the Oz language and its execution model. Oz has deep roots in the logic programming and concurrent logic programming communities. It is illuminating to show these connections. Section 6.5 presents Distributed Oz and its architecture, and explains how it separates functionality from distribution structure. The four protocols are highlighted, namely distributed logic variables, lazy replication of object-records, mobility of object-state, and distributed garbage collection. Finally, Sections 6.6, 6.7, and 6.8 discuss open computing, failure detection and handling, and resource control and security. These three sections are more speculative than the others since they describe parts of the system that are still under development.

6.3 Shared graphic editor

Writing an efficient distributed application can be much simplified by separating the functionality from the distribution structure. We have substantiated this claim by designing and implementing a prototype shared graphic editor, an application which is useful in a collaborative work environment. The editor is seen by an arbitrary number

[6] A fortiori, correctness and termination for nonreactive applications.

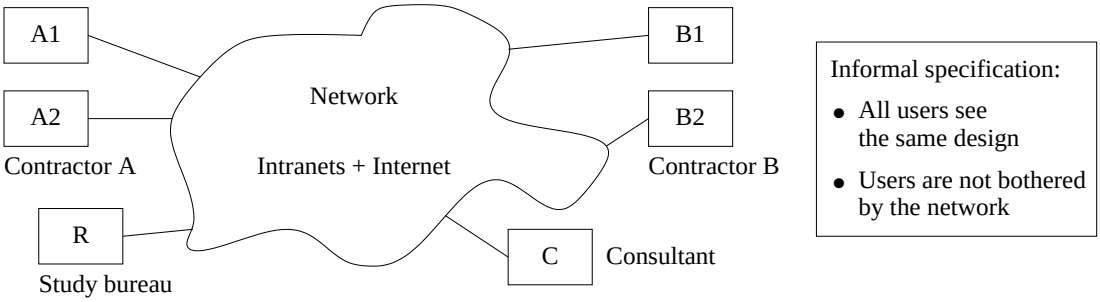


Figure 6.2: A shared graphic editor

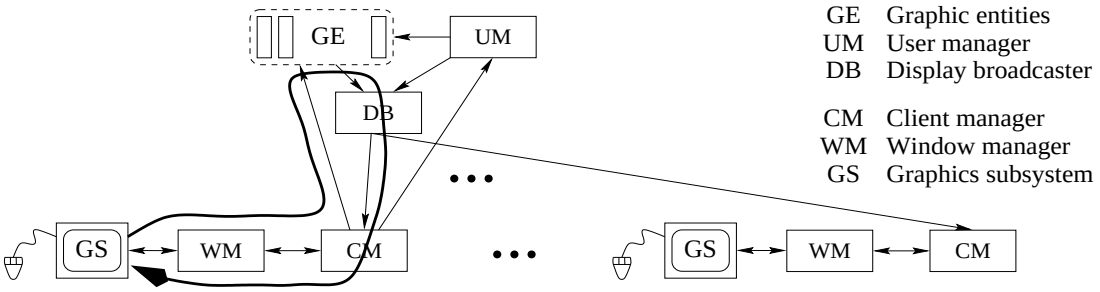


Figure 6.3: Logical architecture of the graphic editor

of users. We wish the editor to behave like a shared virtual environment. This implies the following set of requirements (see Figure 6.2). We require that all users be able to make updates to the drawing at any time, that each user sees his or her own updates without any noticeable delays, and that updates must be visible to all users in real time. Furthermore, we require that the same graphic entity can be updated by multiple users. This is useful in a collaborative CAD environment when editing complex graphic designs. Finally, we require that all updates are sequentially consistent, i.e., each user has exactly the same view of the drawing. The last two requirements is what makes the application interesting. Using IP multicast to update each user's visual representation, as is done for example in the LBL Whiteboard application, [7] does not satisfy the last two requirements.

6.3.1 Logical architecture

Figure 6.3 gives the logical architecture of our prototype. No assumptions are made about the distribution structure. The drawing state is represented as a set of objects. These objects denote graphic entities such as geometric shapes and freehand drawing

[7] Available at `http://mice.ed.ac.uk/mice/archive`.

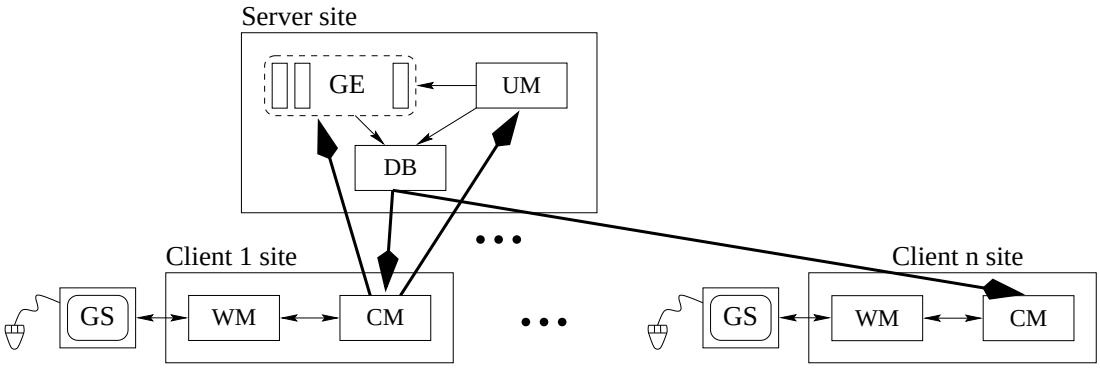


Figure 6.4: Editor with client-server structure

pads. When a user updates the drawing, either a new object is created or a message is sent to modify the state of an existing object. The object then posts the update to a display broadcaster. The broadcaster sends the update to all users so they can update their displays. The execution path from user input to display update is shown by the heavy curved line. The users see a shared stream, which guarantees sequential consistency.

New users can connect themselves to the editor at any time using the open computing ability of Distributed Oz. The mechanism is based on "tickets", which are simply text strings (see Section 6.6). Any Oz process that knows the ticket can obtain a reference to the language entity. The graphic editor creates a ticket for the User Manager object, which is responsible for adding new users. A new user is added by using the ticket to get a reference to the User Manager. The two computations then reference the same object. This transparently opens a connection between two sites in the two computations. From that point onward, the computation space is shared. When there are no more references between two sites in a computation, then the connection between them is closed by the garbage collector. Computations can therefore connect and disconnect seamlessly.

6.3.2 Client-server structure

To realize the design, we have to specify its distribution structure. Figure 6.4 shows one possibility: a client-server structure. All objects are stationary. They are partitioned among a server site and one site per user. This satisfies all requirements except performance. It works well on low-latency networks such as LANs, but performance is poor when a user far from the server tries to draw freehand sketches or any other graphic entity that needs continuous feedback. This is because a freehand sketch consists of many small line segments being drawn in a short time. In our implementation, up to 30 motion events per second are sent from the graphics subsystem to the Oz process. Each line segment requires updating the drawing pad state and sending this update to

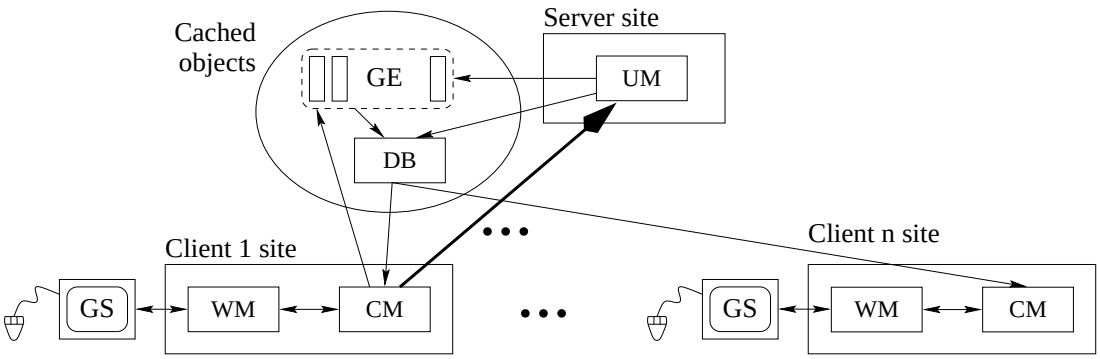


Figure 6.5: Editor with cached graphic state

all users. If the state is remote, then the latency for one update is often several hundred milliseconds or more, with a large variance.

6.3.3 Cached graphic state

To solve the latency problem, we change the distribution structure (see Figure 6.5). We refine the design to represent the graphic state and the display broadcaster as freely mobile ("cached") objects rather than stationary objects. The effect of this refinement is that parts of the graphic state are cached at sites that modify them. Implementing the refinement requires changing some of the calls that create new objects. In all, less than 10 lines of code out of 500 have to be changed. With these changes, freehand sketches do not need any network operations to update the local display, so performance is satisfactory. Remote users see the sketch being made in real time, with a delay equal to the network latency. How is this magic accomplished? It is simple: whenever an object is invoked on a site, then the mobile state protocol first makes the object's state pointer local to the site (see Section 6.5.4). The object invocation is therefore a local operation.

6.3.4 Push objects and transaction objects

More refined editor designs can take advantage of additional distribution behaviors of objects. For example, the design with cached objects suffers from two problems:

- Users who simultaneously modify different graphic entities will interfere with each other through the display broadcaster. The latter will bounce between user sites, causing delays in updating the displays. This problem can be solved by using a *push object*, which multicasts state updates to all sites that reference the object. One possibility is to make the display broadcaster into a push object, thus maintaining sequential consistency while taking advantage of a multicast

network protocol. Another possibility is to make each graphic entity into a push object. In this case, the users may see inconsistent drawings.

- If a user wishes to modify a graphic entity, there is an initial delay while the graphic entity's state is cached on the user site. This problem can be solved by using a *transaction object*, which does the state update locally, while requesting a global lock on the object. The state update will eventually be confirmed or rejected.

Both push and transaction objects maintain consistency of object updates: the object is defined by a sequence of states. It follows that there is still one graphic state and updates to it are sequentially consistent. The editor therefore still supports collaborative design. What changes is how the state sequence is seen and how it is created.

Updating the editor to use either or both of these object types may require changing its specification or logical architecture. For example, the specification may have to be relaxed slightly, temporarily allowing incorrect views. This illustrates the limits of network-transparent programming. It is not possible in general to indefinitely improve the performance of a given specification and logical architecture by changing the distribution structure. At some point, one or both of the specification and architecture must be changed.

6.3.5 Final comments

Designing the shared graphic editor illustrates the two-part approach for building applications in Distributed Oz. First, build and test the application using stationary objects. Second, reduce latency by carefully selecting a few objects and changing their mobility behavior. Because of transparency, this can be done with quite minor changes to the code of the application itself. This can give good results in many cases. To obtain the very best performance, however, it may be necessary to change the application's specification or architecture.

In both the stationary and mobile designs, fault tolerance is a separate issue that must be taken into account explicitly. It can be done by recording on a reliable site a log of all display events. Crashed users disappear, and new users are sent a compressed version of the log. Primitives for fault tolerance are given in Section 6.7.

In general, mobile objects are useful both for fine-grained mobility (caching of object state) as well as coarse-grained mobility (explicit transfer of groups of objects). The key ability that the system must provide is transparent control of mobility, i.e., control that is independent of the object's functionality. Sections 6.4.2 and 6.5 explain briefly how this is done in Distributed Oz. A full explanation is given in [132].

6.4 Oz

Oz is a rich language built from a small set of powerful ideas. This section attempts to situate Oz among its peers. We summarize its programming model and we compare it

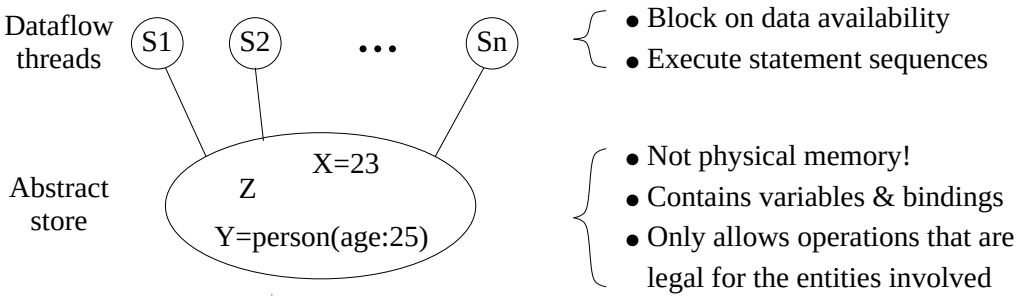


Figure 6.6: Computation model of OPM

S ::=	$S\ S$	Sequence
	X=`f(`l_1`:`Y_1 `...` l_n`:`Y_n`)` |	Value
	X=`<number>` | X=`<atom>` | `{NewName` X`}`	
	`local` X_1 `...` X_n **`in`** S **`end`** | X=Y	Variable
	`proc` `{`X Y_1 `...` Y_n`}` S **`end`** | `{`X Y_1 `...` Y_n`}`	Procedure
	`{NewCell` Y X`}` | `{Exchange` X Y Z`}` | `{Access` X Y`}`	State
	`case` X`==`Y **`then`** S **`else`** S **`end`**	Conditional
	`thread` S **`end`** | `{GetThreadId` X`}`	Thread
	`try` S **`catch`** X **`then`** S **`end`** | **`raise`** X **`end`**	Exception

Figure 6.7: Kernel language of OPM

with Prolog and with concurrent logic languages.

The roots of Oz are in concurrent and constraint logic programming. The goal of the Oz project is to provide a firm foundation for *all* facets of computation, not just for a declarative subset. The semantics should be fully defined and bring the operational aspects out into the open. For example, concurrency and stateful execution make it easy to write programs that interact with the external world [58]. True higher-orderness results in compact, modular programs [4]. First-class computation spaces allow to program inference engines within the system. For example, it is easy to program multiple concurrent first-class Prolog top levels, each with its own search strategy [111].

Section 6.4.1 summarizes the Oz programming model, including the kernel languages and the abstractions built on top of it. Section 6.4.2 illustrates Oz by means of a nontrivial example, namely the implementation of remote method invocation. Section 6.4.3 compares Oz and Prolog. Finally, Section 6.4.4 gives the history of Oz from a concurrent logic programming viewpoint.

6.4.1 The Oz programming model

The basic computation model is an abstract store observed by dataflow threads (see Figure 6.6). A thread executes a sequence of statements and blocks on the availability of data. The store is not physical memory. It only allows operations that are legal for the entities involved, i.e., no type casting or address calculation. The store has three compartments: the constraint store, containing variables and their bindings, the procedure store, containing procedure definitions, and the cell store, containing mutable pointers ("cells"). The constraint and procedure stores are monotonic, i.e., information can only be added to them, not changed or removed. Threads block on availability of data in the constraint store.

The threads execute a kernel language called Oz Programming Model (OPM) [116]. We briefly describe the OPM constructs as given in Figure 8.3. Statement sequences are reduced sequentially inside a thread. Values (records, numbers, etc.) are introduced explicitly and can be equated to variables. All variables are logic variables, declared in an explicit scope defined by the **`local`** construct. Procedures are defined at run-time with the **`proc`** construct and referred to by a variable. Procedure applications block until their first argument refers to a procedure. State is created explicitly by `NewCell`, which creates a *cell*, an updatable pointer into the constraint store. Cells are updated by `Exchange` and read by `Access`. Conditionals use the keyword **`case`** and block until the condition is true or false in the constraint store.[8] Threads are created explicitly with the **`thread`** construct and have their own identifier. Exception handling is dynamically scoped and uses the **`try`** and **`raise`** constructs.

Full Oz is defined by transforming all its statements into this basic model. Full Oz supports idioms such as objects, classes, reentrant locks, and ports [116, 132]. The system implements them efficiently while respecting their definitions. We define the essence of these idioms as follows. For clarity, we have made small conceptual simplifications. Full definitions may be found in [51].

- **Object**. An object is essentially a one-argument procedure `{Obj M}` that references a cell, which is hidden by lexical scoping. The cell holds the object's state. The argument `M` indexes into the method table. A method is a procedure that is given the message and the object state, and calculates the new state.

- **Class**. A class is essentially a record that contains the method table and attribute names. When a class is defined, multiple inheritance conflicts are resolved to build its method table. Unlike Java, classes in Oz are pure values, i.e., they are stateless.

- **Reentrant lock**. A reentrant lock is essentially a one-argument procedure `{Lck P}` used for explicit mutual exclusion, e.g., of method bodies in objects used concurrently. `P` is a zero-argument procedure defining the critical section. Reentrant means that the same thread is allowed to reenter the lock. Calls to the lock

[8] The keyword **`if`** is reserved for constraint applications.

```
proc {NewStationary Class Init ?StatObj}
   Obj={New Class Init}
   S P={NewPort S}
   N={NewName}
in
   thread
     {ForAll S
      proc {$ M#R}
         thread
            try {Obj M} R=N
            catch E then R=E end
         end
      end}
   end
   proc {StatObj M}
      R in
      {Send P M#R}
      case R==N then skip
      else raise R end
      end
   end
end
```

Figure 6.8: RMI part 1: Create a stationary object from any class

may therefore be nested. The lock is released automatically if the thread in the body terminates or raises an exception that escapes the lock body.

- **Port**. A port is an asynchronous channel that supports many-to-one communication. A port `P` encapsulates a stream `S`. A stream is a list with unbound tail. The operation `{Send P M}` adds `M` to the end of `S`. Successive sends from the same thread appear in the order they were sent.

6.4.2 Oz by example

It is not the purpose of this article to give a complete exposition of Oz. Instead, we present Oz by means of a nontrivial example program that is interesting in its own right. We show how to implement active objects in Oz, and as a corollary, we show that the same program implements remote method invocation in Distributed Oz. An active object is an object with an associated thread (or process), much like an actor or concurrent logic process. Invoking a method in an active object is done by explicitly sending a message to the associated thread. As we will see, this kind of object has a well-defined distribution behavior in Distributed Oz. Because threads are stationary in

```
class Counter
   attr i
   meth init i<-0 end
   meth inc i<- @i+1 end
   meth get(X) X=@i end
   meth error raise some_error end end
end

Obj={NewStationary Counter init}
{Obj inc}
{Obj inc}
{Print {Obj get($)}}
try {Obj error} catch X then {Print X} end
```

Figure 6.9: RMI part 2: A stationary counter object

Distributed Oz, the objects also are stationary and reside on their creation site. Invoking the object from a remote site behaves exactly like a remote method invocation.

In Distributed Oz, objects are mobile by default and will execute on the invoking site, inside the invoking thread. This is implemented by a lightweight mobility protocol that serializes the path of the object's state pointer among the invoking sites (see Section 6.5). One way to make an object stationary is to wrap it inside a port and create a thread that invokes the object with messages read from the port's stream. The object is accessed only from this thread, so the object is stationary.

Figure 6.8 defines the procedure `NewStationary` that implements stationary objects by wrapping them inside a port. It takes a class `Class` and initialization message `Init`, and returns a procedure `StatObj`. The "?" is a comment that denotes an output argument. Inside `NewStationary`, an object `Obj` is created, as well as a port `P` and its associated stream `S`. A thread is created that serves each message appearing on `S`. This is done using the higher-order procedure `{ForAll S Proc}` where `Proc` is a one-argument procedure. The thread waits until a message `M#R` appears on the stream `S` and then executes the procedure call `{Proc M#R}`. The procedure starts a thread that invokes the object with `{Obj M}` and binds `R` either to a unique name `N` denoting normal execution or to an exception `E`. The use of the lexically-scoped new name `N` avoids conflicts with existing exceptions. Let us now consider the procedure `StatObj`. A thread executing `StatObj` sends on the port `P` the pair `M#R` where `M` is the message and `R` is a logic variable for the answer. It suspends on `R` until the corresponding method is executed successfully or an exception is returned. In the latter case the exception is reraised in the thread executing `StatObj`.

We see that Oz allows the programmer to provide generic abstractions that can be used later without concern for their implementation. It is not necessary to understand `NewStationary` in order to use it. This is because the objects it creates have the same

	SICStus Prolog	Oz
Constraints	Incremental solver with tell	Incremental solver with ask, tell
Control	Backtracking and coroutining	Explicit dataflow threads, encapsulated search
Higher-order	Call, assert	First-class procedures with lexical scoping
State	Objects, mutables, assert	Objects, cells

Table 6.1: Oz and Prolog

Oz semantics as objects created by the standard procedure `New`.

Figure 6.9 defines a `Counter` class, creates a stationary instance, `Obj`, and sends several messages to `Obj`. Whether `Obj` is created by `NewStationary` or `New`, its language behavior is the same. The `Counter` class does not have any ancestors, therefore no inheritance declaration appears. Each instance of `Counter` has one attribute `i` and four methods. An attribute is a mutable part of the object state that can be accessed and modified from within a method. A method is defined by a method head, which is a record, and a method body, which is a statement. Dynamic binding is supported through the use of **`self`** inside a method body. Accessing the value of an attribute is done by the operator "`@`". Assigning a new value to an attribute is done by the operator "`<-`". Therefore, the method `init` initializes `i` to `0`, the method `inc` increments `i`, the method `get` gets the current value of `i`, and the method `error` raises the somewhat unusual exception `some_error`. Oz has syntactic support for embedding statements in expressions. A statement can be used as an expression by using a "`$`" to mark the result. Therefore `{Print {Obj get($)}}` is equivalent to **`local`** `X` **`in`** `{Obj get(X)} {Print X}` **`end`**.

6.4.3 Oz and Prolog

There is a strong sense in which Oz is a successor to Prolog (see Table 6.1). The Oz system can be used for many of the tasks for which Prolog and constraint logic programming are used today [88, 111, 67, 113]. Like Prolog, Oz has a declarative subset. Like Prolog, Oz has been generalized to arbitrary constraint systems (currently implemented are finite domains and open feature structures). Oz is fully defined and has an efficient implementation competitive with the best emulated Prolog systems [58, 91, 127]. Even though Oz has much in common with Prolog, it is not a Prolog superset. Oz does not have Prolog's reflective syntax (i.e., data and programs have the same syntax), nor does it have the meta-programming facilities (like `call/1`, `assert/1`) or the user-definable syntax (operator declarations).

The foundation of Prolog's success is the high abstraction level of its declarative subset, namely first-order Horn clause logic with SLDNF resolution [84]. What's missing from Prolog is that little attempt is made to give the same foundation to anything *outside* the declarative subset. Two decades of research have resulted in a solid un-

	Concurrent logic programming	Oz
Constraints	None, except in AKL	Incremental solver with ask, tell
Control	Fine-grained concurrency	Explicit dataflow threads, encapsulated search
Higher-order	Restricted	First-class procedures with lexical scoping
State	Stream-based objects	Objects, cells

Table 6.2: Oz and concurrent logic programming

derstanding of the declarative subset and only a partial understanding of the rest.[9] This results in two main flaws of Prolog. First, the operational aspects are too deeply intertwined with the declarative. The control is naive (depth-first search) and eager. The interactive top level has a special status: it is lazy, but unfortunately inaccessible to programs. It is lazy because new solutions are calculated upon user request. It is inaccessible to programs, i.e., a program cannot internally set up a query and request solutions lazily. To provide a top level within a program requires programming a meta-interpreter, thus losing an order of magnitude in efficiency. Second, to express anything beyond the declarative subset requires ad hoc primitives that are limited and do not always do the right thing. The `freeze/2` provides coroutining as a limited form of concurrency. The `call/1` and `setof/3` provide only a limited form of higher-orderness. All these problems are solved in Oz.

6.4.4 Oz and concurrent logic programming

Oz is the latest in a long line of concurrent logic languages. Table 6.2 compares Oz with concurrent logic programming languages. First experiments with concurrency were done in the venerable IC-Prolog system where coroutining was used to simulate concurrent processes. This led to the Parlog language and Concurrent Prolog. The advent of GHC simplified concurrent logic programming considerably by introducing the notion of *quiet guards*. A clause matching a goal will fire only if the guard is entailed by the constraint store. This formulation and its theoretical underpinning were pioneered by the work of Maher and Saraswat as they gave a solid foundation to concurrent logic programming [86, 107]. On the practical side, the flat versions of Concurrent Prolog and GHC, called FCP and FGHC respectively, were the focus of much work [66, 114]. The KL1 language, derived from FGHC, was implemented in the high-performance KLIC system. This system runs on sequential, parallel, and distributed machines [44]. A number of implementation techniques in the current Distributed Oz system have been borrowed from KLIC, notably the distributed garbage collection algorithm.

An important subsequent development was AKL (Andorra Kernel Language) [69],

[9] The non-declarative aspect has received some attention, e.g., [95, 100, 9].

Kind of entity	Protocol		Entity
Stateless	Replication	Eager	record, procedure, class
		Lazy	object-record
Single assignment	Binding	Eager	logic variable
		Lazy	logic variable
Stateful	Localization	Mobile	cell, object-state
		Stationary	port, thread

Table 6.3: Semantics of Distributed Oz

which added explicit state in the form of ports and provided the first synthesis of concurrent and constraint logic programming. AKL encapsulates search by using nested computation spaces. A computation space is a constraint store with its associated goals. Search is done by allowing procedures to be defined by a sequence of don't-know guarded clauses. These definitions denote disjunctions. When local propagation cannot choose between different disjuncts, then the program is free to try them by cloning the computation space. The initial Oz system, Oz 1, was largely derived from AKL, but added the notions of higher-order procedures, more controllable search by making computation spaces first class, compositional syntax, and the cell primitive for mutable state. Concurrency in Oz 1 is fine-grained. When a statement suspends, a new thread is created that contains only the suspended statement. The main thread is not suspended but continues with the next statement.

All concurrent logic languages up to and including Oz 1 were designed for fine-grained concurrency and implicit exploitation of parallelism. The current Oz language, Oz 2, abandons this model in favor of explicit control over concurrency by means of a thread creation construct. Thread suspension and resumption is still based on dataflow using logic variables. Our experience shows that explicit concurrency makes it easier for the user to control application resources. It allows the language to have an efficient and expressive object-oriented model without sequential state threading within method definitions. It also allows easy incorporation of a conventional exception handling construct into the language, and last but not least a simple debugging model. In the current Oz system concurrency is used mostly to model logical concurrency in the application rather than to increase potential parallelism.

6.5 Distributed Oz

Distributed Oz has the same language semantics as Oz. Distributed Oz separates application functionality from distribution structure by defining a distributed semantics for all language entities [132, 131, 53, 55]. The distributed semantics extends the language semantics to take into account the notion of site. It defines the network operations invoked when a computation is partitioned on multiple sites. We classify the language entities into three basic types (see Table 6.3):

- Stateless entities are replicated eagerly (records, procedures, classes) or lazily

(object-record).

- Single assignment entities (logic variables) are bound eagerly or lazily [53].
- Stateful entities are localized and are either mobile by default (cell, object-state) or stationary by default (port, thread) [132]. What moves is not the state, but the site that has the right to create the next state. We say that this site has the *state pointer*.[10]

For each of these entities, network operations[11] are predictable, which gives the programmer the ability to manage network communications. In the rest of this section, we present the four distributed algorithms used to implement the language entities. Section 8.2 introduces the concept of *access structure*, which models a language entity that is accessible from more than one site. The distributed behavior of a language entity is defined as a protocol between the nodes of its access structure, i.e., as a distributed algorithm. Sections 6.5.2 explains the uses of distributed logic variables and shows how to bind them with a *variable binding* protocol. Sections 6.5.3 and 6.5.4 show how to build mobile objects that have predictable network behavior by using a *lazy replication* protocol for the object-record (explained in Section 6.5.3) and a *mobile state* protocol for the object-state (explained in Section 6.5.4). The network behavior of logic variables and objects highlights most clearly the design philosophy of Distributed Oz. Finally, Section 6.5.5 explains the distributed garbage collection algorithm, which underlies the management of access structures.

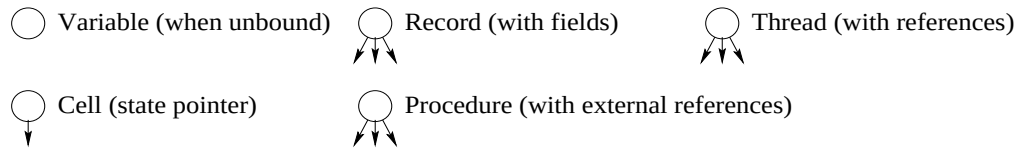


Figure 6.10: Language entities as nodes in a graph

6.5.1 The distribution graph

We model distributed execution in a simple but precise manner using the concept of *distribution graph*. We obtain the distribution graph in two steps from an arbitrary execution state of the system. The first step is independent of distribution. We model the execution state by a graph, called *language graph*, in which each language entity except for an object corresponds to one node (see Figure 6.10). Objects are compound entities and are explained in Section 6.5.3.

In the second step, we introduce the notion of *site*. Assume a finite set of sites and annotate each node by its site (see Figure 6.11). If a node, e.g., `N2`, is referenced by at least one node on another site, then map it to a *set* of nodes, e.g., {`P1`,`P2`,`P3`,`M`}.

[10] In [132] it is called the *content-edge*.

[11] In terms of the number of network hops.

This set is called the *access structure* of the original node. An access structure consists of one *proxy node* `Pi` for each site that referenced the original node and one *manager node* `M` for the whole structure. The resulting graph, containing both local nodes and access structures where necessary, is called the *distribution graph*. Most of the example protocol executions in this article use this notation.

Each access structure is given a global address that is unique system-wide. The global address encodes various pieces of information including the manager site. Proxy nodes are uniquely identified by pairs (global address,site). On each site, the global address indexes into a table that refers to the proxy. This allows to enforce the invariant that each site has at most one proxy. Messages are sent between nodes in access structures. In terms of sites, a message is sent from the source node's site to the destination node's site. In the message body, all references are to nodes on the destination site. These nodes are identified by the global addresses of their access structures. When the message arrives, the nodes are looked up in the site table.

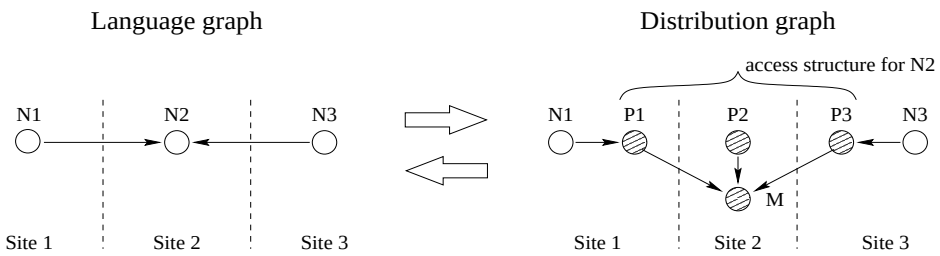


Figure 6.11: Access structure in the distribution graph

Procedures and other values (records and numbers, etc.) are copied eagerly, i.e., they never result in an access structure.[12] A procedure is only sent once to any site[13] and has only one copy on the site. A procedure consists of a closure and a code block, each of which is given a global address. Messages contain only the global addresses, and upon arrival the missing code blocks and closures are requested immediately.

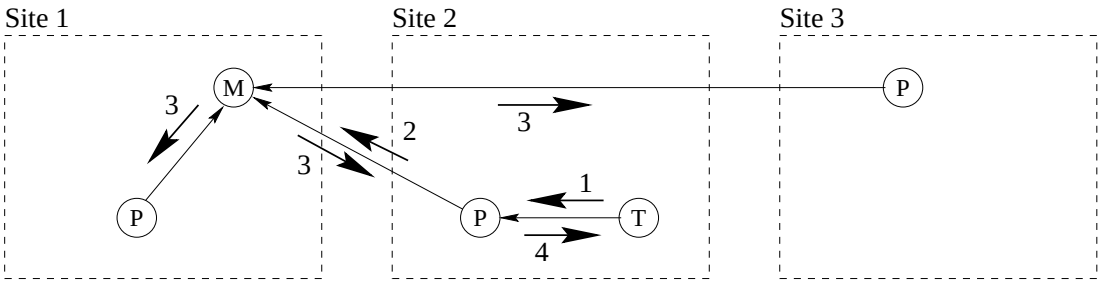
6.5.2 Distributed logic variables

Logic variables express dependencies between computations without imposing an execution order. This property can be exploited in distributed computing:

- Two basic problems in distributed computing are latency tolerance and third-party independence. Using logic variables instead of explicit message passing can improve these two aspects of an application with little programming effort.
- Using logic variables, common distributed programming idioms can be expressed in a network-transparent manner that results in optimal or near-optimal message traffic.

[12] In [132] there is a variant design in which objects are procedures and all procedures are copied lazily.

[13] Unless a garbage collection removes it.



1 Thread initiates binding and blocks
2 Proxy requests binding
3 Manager grants binding & multicasts to all proxies
4 Proxy informs thread, allowing thread to continue

Figure 6.12: Binding a logic variable

These benefits are realized due to a practical distributed algorithm for rational tree unification, which is used to bind logic variables [53]. The algorithm is efficiently implemented in the Distributed Oz system as two parts: a local algorithm and a distributed algorithm. Most of the work of unification is done locally. The distributed algorithm does only variable binding. We briefly describe it here.

The two basic operations on logic variables are binding and waiting until bound. A logic variable `X` can be bound to a data structure or to another variable. The algorithm is the same in both cases. If many bindings to `X` are initiated concurrently (from one or more sites), then only one will succeed. The other bindings are then retried with the entity to which `X` is bound. By default, binding is *eager*. That is, the new value is immediately sent to all sites that know about `X`. This means that a bound variable is guaranteed to eventually disappear from the system.

We illustrate the binding algorithm with an example. In the distribution graph, a logic variable shows up as an access structure. Figure 6.12 shows a variable that exists on three sites. A thread on site 2 initiates a binding of the variable by informing its proxy (message 1) and then blocking. The proxy asks the manager to bind the variable (message 2). The manager informs all proxies of the binding (message 3), thus binding the variable eagerly. When a proxy receives the binding, it informs all waiting threads (message 4). The threads then continue execution.

Logic variables can have different distributed behaviors, as long as network transparency is satisfied in each case. A logic variable is *eager* by default. This gives maximal latency tolerance and third-party independence. However, this may cause the binding to be sent to sites that do not need it. We say that a logic variable is *lazy* if its value is only sent to a site when the site requests it (e.g., when a thread needs the value). A lazy variable has better message complexity, i.e., fewer messages are used. In some cases, e.g., implementing barrier synchronization using a short-circuit technique, lazy variables are preferable. Eager and lazy variables obey the same distributed

unification algorithm, differing only in the scheduling of one reduction rule [53]. Distributed Oz currently only implements eager variables; with a minor change it can do both. A programmer annotation can then decide whether a variable is eager or lazy.

```
class Account
   attr bal:0
   meth trans(Amt)
      bal<- @bal+Amt
   end
   meth getBal(B)
      B = @bal
   end
end

A={New Account trans(100)}
```

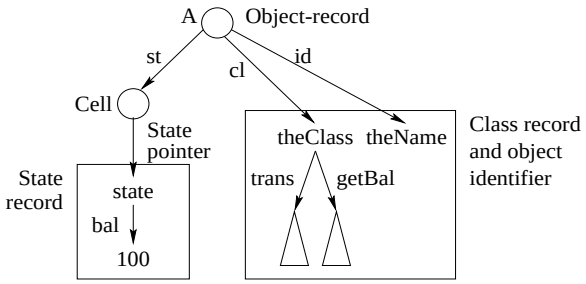


Figure 6.13: An object with one attribute and two methods

6.5.3 Mobile objects

Objects in Distributed Oz obey a lightweight object migration protocol that preserves centralized object semantics and allows for precise prediction of network behavior. Existing systems with mobile objects do not use such an algorithm. They move the objects by creating a chain of forwarding references [90, 72, 21]. This chain is short-circuited when a message is sent or after a given time delay. This gives good average-case number of network hops when moving an object, but very bad worst-case number of hops. A design principle of Distributed Oz is for third-party dependencies to disappear quickly. Using chains is therefore unacceptable. Instead, we have designed the mobility protocol presented here, which has a much-improved worst-case behavior.

In the distribution graph, an object shows up as a compound entity consisting of an object-record, a class record containing procedures (the methods), a cell (containing the state pointer), and a record containing the object-state. The distributed behavior of the object is derived from the behavior of its parts. Figure 6.13 shows an object `A` that has one attribute, `bal`, and two methods, `trans` and `getBal`. The object is represented as an object-record with three fields. The `st` field contains a cell, whose state pointer refers to the object's state record. The `cl` field contains the class record, which contains the procedures `trans` and `getBal` that implement the methods. The `id` field contains the object's unique identifier `theName`. The object-record and the class record cannot be changed. However, by giving a new content to the cell (i.e., updating the state pointer), the object-state can be updated.

Figure 6.14 shows an object `A` that is local to Site 1. There are no references to `A` from any other sites. Figure 6.15 shows an object `A` with one remote reference. The object is now part of an access structure whose manager is on Site 1 and that has one proxy on Site 2. A local object `A` is transformed to a global (i.e., remotely-referenced) object when a message referencing `A` leaves Site 1. A manager node `Ma` is created on Site 1 when the message leaves. When a message referencing `A` arrives on Site 2, then a proxy node `Pa2` is created there.

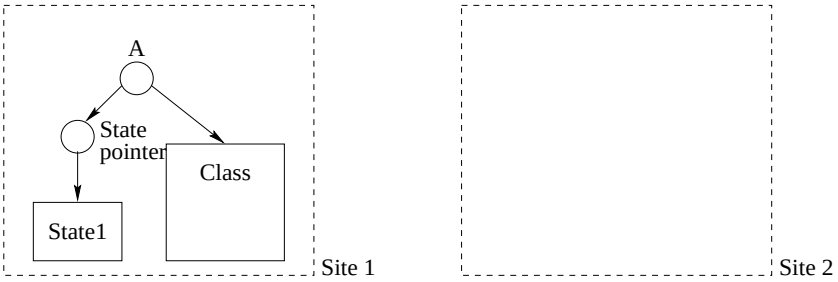


Figure 6.14: A local object

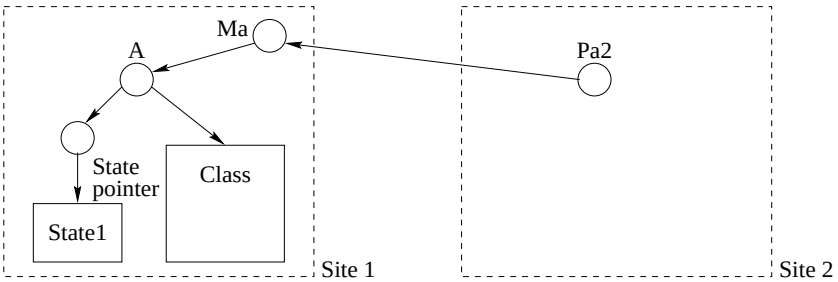


Figure 6.15: A global object with one remote reference

Figure 6.16 shows what happens when thread `T` invokes `A` from Site 2. At first, only the proxy `Pa2` is present on Site 2, not the object itself. The proxy asks its manager for a copy of the object-record. This causes an access structure to be created for the cell, with a manager `Mc` and one proxy `Pc1`. The class record is copied eagerly and does not have a unique global address. A message containing the class record and a cell proxy is sent to Site 2. The object's state remains on Site 1.

Figure 6.17 shows what happens when the message arrives. A second proxy `Pc2` is created for the cell. The class record is copied to Site 2 and proxy `Pa2` becomes the object-record `A`. The site table now refers to the object-record. The mobile state protocol (see Section 6.5.4) then atomically transfers the cell's state pointer to Site 2. Because of the site table, any further messages to Site 2 containing references to the object will immediately refer to the local copy of the object-record, without requiring any additional network operations.

Figure 6.18 shows what happens after the state pointer is transferred to Site 2. The new state, `State2`, is created on Site 2 and will contain the updated object-state after the method finishes. The old state, `State1`, may continue to exist on Site 1 but the state pointer no longer points to it.

Figure 6.19 shows what happens if Site 1 invokes the object again. The state pointer is transferred back to Site 1. The new state, `State3`, is created on Site 1 and will contain the updated object-state after the method finishes. The old state, `State2`, may continue to exist on Site 2 but the state pointer no longer points to it.

There are several interesting things going on here. First, the object is always ex-

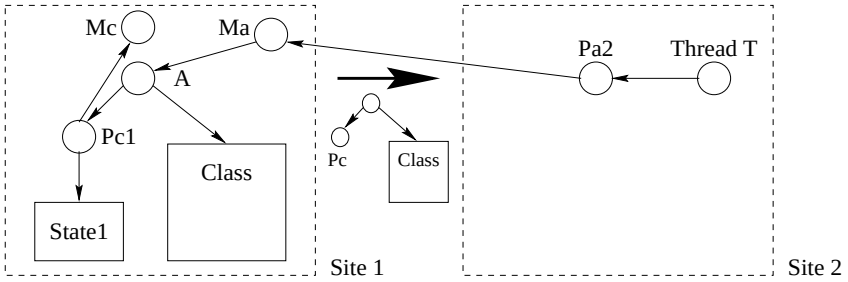


Figure 6.16: The object is invoked remotely (1)

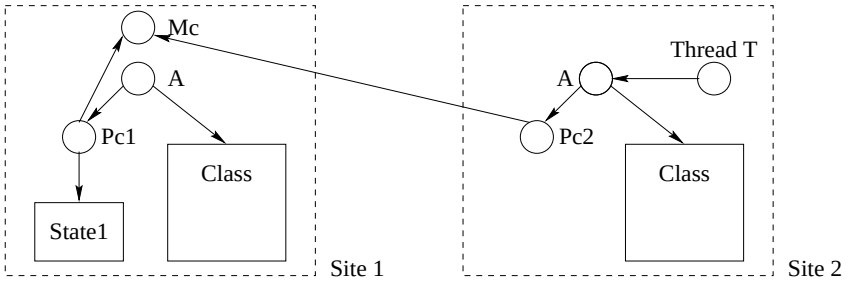


Figure 6.17: The object is invoked remotely (2)

ecuted locally. The cell's state pointer is always localized before the method starts executing and it is guaranteed to stay local during the method execution while the object is locked. Second, the class code is only transferred once to any site. Only the state pointer is moved around after the first transfer. This makes object mobility very lightweight. Third, all requests for the object are serialized by the cell's manager node. This simplifies the protocol but introduces a dependency on the manager site. A more complicated protocol (not shown here) can remove this dependency [132].

6.5.4 Mobile state

The freely mobile objects shown in Section 6.5.3 are composite entities that use several distributed algorithms. The object-record is copied once lazily (when the object is first invoked), the methods are copied along with it, and the object's state pointer is moved between sites that request it. At all times, the state pointer of the object's cell access structure exists at exactly one proxy, or is in transit between two proxies. The protocol that moves the state pointer, the *mobile state* protocol, is particularly interesting. This protocol must guarantee consistency between consecutive states. If the consecutive states are on different sites, this requires an atomic transfer of the state pointer between the sites. A site that wants the state pointer requests it from the cell manager, and the latter sends a forwarding command to the site that has the state pointer. Therefore the manager needs to store only one piece of information, namely the site containing the state pointer [132].

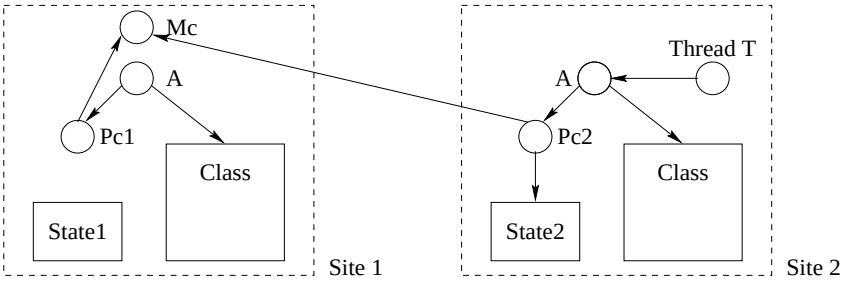


Figure 6.18: The object is invoked remotely (3)

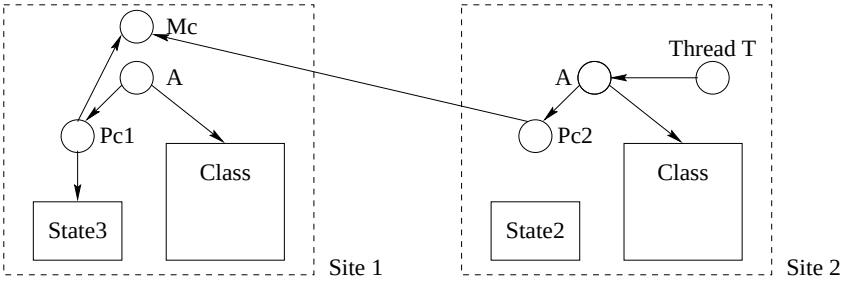


Figure 6.19: The object moves back to Site 1

Figure 6.20 shows a cell `C` referenced from two sites. The cell's state pointer is on Site 1 and Site 2 requests it when thread `T` does the operation {`Exchange C X Y`}. It suffices to know that the exchange is an atomic swap that sets the new state to `Y` (i.e., to `State2`) and initiates a binding of `X` to the old state `State1`.

Figure 6.21 shows (a) proxy `Pc2` requesting the state pointer by sending a `Get` message to manager `Mc`, and (b) the manager sending a `Forward` message to the proxy that has (or will eventually have) the state pointer, namely `Pc1`. Therefore the manager can accept another request immediately; it does not need to wait until the state pointer's transfer is complete.

Figure 6.22 shows `Pc1` sending to `Pc2` a `Content` message containing the old state, `State1`. The old state may still exist on Site 1 but `Pc1` no longer has a pointer to it. Figure 6.23 shows the final situation. `Pc2` has the state pointer, which points to `State2`. `X` is bound to `State1`.

This protocol provides a predictable network behavior. There are a maximum of three network hops for the state pointer to change sites; only two if the manager is on the source or destination site; zero if the state pointer is on the requesting site. The protocol maintains sequential consistency, that is, cell exchanges (updates of the state pointer) are done in a globally consistent order.

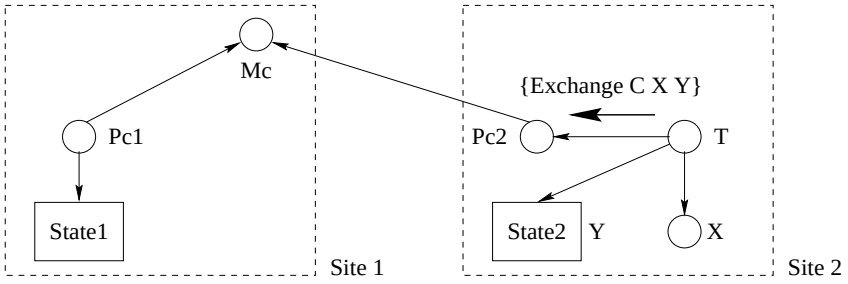


Figure 6.20: A cell referenced from two sites

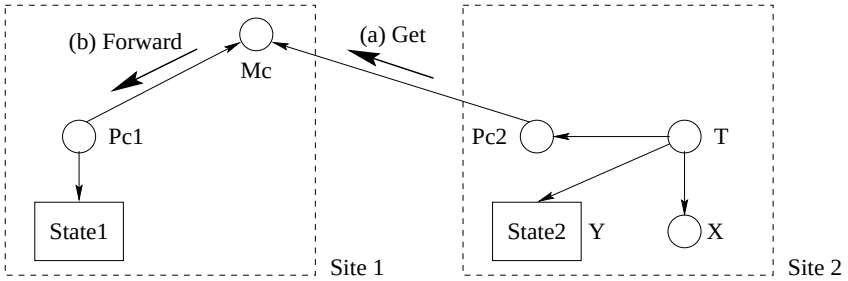


Figure 6.21: (a) Site 2 requests the state pointer; (b) Site 1 is asked to forward it

6.5.5 Distributed garbage collection

Access structures are built and managed automatically when language entities become remotely referenced. This happens whenever messages exchanged between nodes on different sites contain references to other nodes. If the reference is to a local node, then the memory management layer converts the local node into an access structure. We say the local node is *globalized*. While the message is in the network, the access structure consists of a manager and one proxy. When the message arrives at the destination site, then a new proxy is created there. Access structures can reduce in size and disappear completely through garbage collection.

Distributed garbage collection is implemented by two cooperating mechanisms: a local garbage collector per site and a distributed credit mechanism to reclaim global addresses. A local garbage collector informs the credit mechanism when a node is no longer referenced on its site. Conversely, the credit mechanism informs the local garbage collector when a node is no longer remotely referenced. Local collectors can be invoked at any time independently of other sites. The roots of local garbage collection are all nodes on its site that are reachable from non-suspended thread nodes or are remotely referenced.

A global address is reclaimed when the node that it refers to is no longer remotely referenced. This is done by the credit mechanism, which is informed by the local garbage collectors. This scheme recovers all garbage except for cross-site cycles. The only cross-site cycles in our system occur between different objects or cells. Since records and procedures are both replicated, cycles between them will be localized to

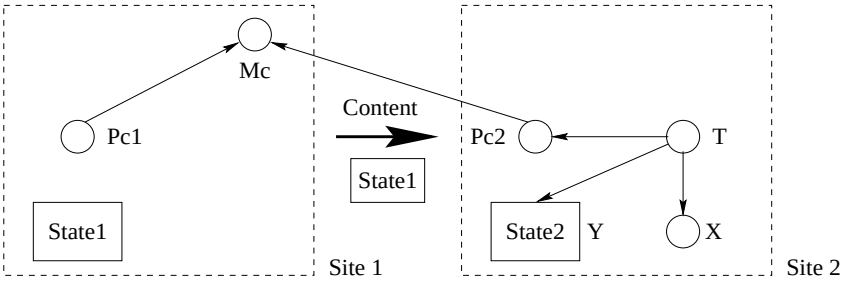


Figure 6.22: Site 1 has sent the state pointer to Site 2

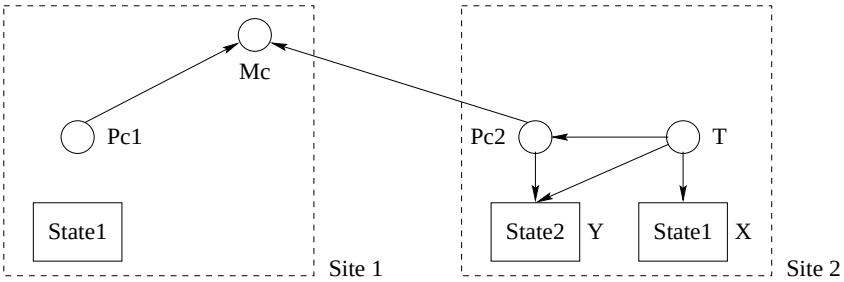


Figure 6.23: Site 2 has the state pointer

single sites. The credit mechanism does not suffer from the memory or network inefficiencies of previous reference-counting schemes [99].

We summarize briefly the basic ideas of the credit mechanism. Each global address is created with an integer (its *debt*) representing the number of *credits* that have been given out to other sites and to messages. Any site or message that contains the global address must have at least one credit for the global address. The creation site is called the *owner*. All other sites are called *borrowers*. A node is remotely referenced if and only if its debt is nonzero.

Initially there are no borrowers, so the owner's debt is zero. The owner lends credits to any site or message that refers to the node and increments its debt each time by the number of credits lent. When a message arrives at a borrower, its credits are added to the credits already present. When a message arrives at the owner, its credits are deducted from the owner's debt. When a borrower no longer locally references a node then all its credits are sent back to the owner. This is done by the local garbage collector. When the owner's debt is zero then the node is only locally referenced, so its global address will be reclaimed.

Consider the case of a cell access structure. The manager site is the owner, and all other sites with cell proxies are borrowers. A proxy disappears when no longer locally referenced. It then sends its credit back to the manager. If the proxy contains the state pointer, then the state pointer is transferred back to the manager site as well. Remark that this removes a cross-site cycle within the cell access structure. When the manager recovers all its credit then it disappears, and the cell becomes a local cell again. When

the local cell has no local references, then it is reclaimed. If the local cell becomes global again (because a message referring to it is sent across the network), then a new manager is created, completely unrelated to the reclaimed one.

6.6 Open computing

We say a distributed system is *open* if independently-running applications can interact in interesting ways [30]. In general, this means that the system must have common ground, in the form of common frameworks or languages, that applications can use to interact. Typical examples are common information formats for exchanging information, common protocols for electronic commerce, etc. As a first requirement, applications must be able to establish connections with computations that have been started independently across the net. A second requirement is that applications should be able to initiate new distributed computations.

6.6.1 Connections and tickets

Distributed Oz uses a ticket-based mechanism to establish connections between independent sites. In the final system, both the tickets and the connections must be implemented in a secure way (see Section 6.8). In this section, we explain the basic mechanism without discussing security issues. One site (called the server site) creates a ticket with which other sites (called client sites) can establish a connection. The ticket is a character string which can be stored and transported through all media that can handle text, e.g., phone lines, electronic mail, paper, and so forth.

The ticket identifies both the server site and the language entity to which a remote reference will be made. Independent connections can be made to different entities on the same site. Establishing a connection has two effects. First, the sites connect by means of a network protocol (e.g., TCP). Second, in the Oz computation space, a reference is created on the client site to a language entity on the server site. The second effect can be implemented by various means, i.e., by passing a zero-argument procedure, by unifying two variables, or by passing a port which is then used to send further values. Once an initial connection is established, then further connections as desired by applications can be built from the programming abstractions available in Oz. For example, it is possible to define a class `C` on one site, pass `C` to another site, define a class `D` inheriting from `C` on that site, and pass `D` back to the original site. This works because Distributed Oz is fully transparent.

Oz features two different types of tickets: one-shot tickets that are valid for establishing a single connection only (one-to-one connections), and many-shot tickets that allow multiple connections to the ticket's server (many-to-one connections). One-to-one connections are useful when connecting to newly-started compute servers (see Section 6.6.2). Many-to-one connections are useful in collaborative applications such as the shared graphic editor of Section 6.3. Multiple users connect to this application in order to contribute to a common design.

We sketch a small example for one-to-one connections:

Server	Client

`STkt={Connection.offer X}`
`{PutOnWebPage STkt}`
`{ProcessData X}`
	`CTkt={QueryUser}`
	`X={Connection.take CTkt}`
	`X=data(···)`

The server offers `X` with the system procedure `Connection.offer`, which is part of the module `Connection`. This procedure takes the offered value and returns a new one-shot ticket `STkt`, which is made available on a Web page. The user reads the page, retrieves the ticket, and types it in at the client site, which puts it in variable `CTkt`. The system procedure `Connection.take` then returns a reference to `X`, which becomes a shared reference between the client and server. In this example, `X` is a shared logic variable. It could have been any language entity, e.g., an object or a port. The client binds `X` and the server reads its value. This passes information from the client to the server.

6.6.2 Remote compute servers

Distributed applications mainly fall into two categories. In the first category, applications involve geographically-distributed resources. For example, in the shared graphic editor of Section 6.3 the users are the distributed resources. In the second category, applications use multiple networked computers to increase computation speed. These applications are often structured as a single master computation that coordinates a set of slave computations. The actual computation is carried out by the slaves.

To support the second category, Oz provides the ability to create remote compute servers, which are accessible as Oz objects. This is implemented using the ticket mechanism. After the compute server has been set up, it can be given tasks to do in the form of procedures. The following example shows one way to use a compute server:

```
S={New RemoteServer init(´wallaby.ps.uni-sb.de´)}
{Print {S run(fun {$} 4+5 end $)}}
```

The remote server site is started on the computer with Internet address `´wallaby.ps.uni-sb.de´`. The `run` method takes a zero-argument function that is executed at the remote server. In the example, two numbers are added and the result `9` is returned to the original site, where it is printed.

Setting up a remote server is done in two steps. Assume that a site wishes to create a remote server and then become a client to the server. First, the potential client creates an independently-running Oz site with the help of the operating system.[14] Second, a connection is established between the potential client and the remote server. This is done by passing a one-shot ticket from the potential client to the server. The server takes a logic variable that has been offered by the client and binds it to a stationary

[14] In the current system, a remote site is started by the Unix remote shell command (`rsh`).

object (see Section 6.4.2). This stationary object is used on the client side to implement the `run` method shown in the example above.

As an application of this idea, we are currently investigating distributed search engines for solving combinatorial constraint problems. First experiments show encouraging speedup. Oz supports the two aspects of distributed search engines in a powerful way: search engines can be easily programmed in Oz [112], and the language supports distributed computing well.

6.7 Failure detection and handling

An application is *fault-tolerant* if it can continue to fulfill its specification despite accidental failures of its components. How can one write such applications in Distributed Oz? The theory of fault-tolerant systems explains how to construct such systems as layers of abstractions [68]. Very little work has been done to integrate these abstractions into a language platform so that (1) a fault-tolerant system can be built within the platform, and (2) the integration is orthogonal to the language entities. Most of the language work has been concentrated in the areas of persistence and transactions, by adding models of these concepts to an existing language. It is possible, however, to support fault tolerance in a much simpler way.

We extend the system to support partial failure of sites and individual language entities, and to detect and handle failure of language entities. We provide the means for the programmer to decide what action to take upon failure. This is done by installing "handlers" or "watchers" on individual language entities (see below). These are invoked when a failure occurs. No irrevocable decision is taken by the system; the handlers and watchers are free to take any course of action. In this way, we intend to build a first fault-tolerant layer using the redundancy that comes from having multiple sites in the system. This gives fault tolerance even in the absence of persistence. More refined fault tolerance based on persistence and transactions will be added later.

6.7.1 The containment principle

Fault tolerance is a property that crosses abstraction boundaries [78]. An example will make this clear. Most existing systems (we include applications) do not handle *time* correctly. What they do is let a lower layer make an irrevocable decision, in the form of a *time-out* that does not let the system continue. Say there is a time-out in a lower layer, for example in the transport layer (TCP) of the network interface. This time-out crosses all abstraction boundaries to appear at the top level, i.e., to the user. Usually, a window is opened asking confirmation to abort the application. The user does not have the possibility to communicate back to the timed-out layer. This greatly limits the flexibility of the system. It should be possible to build a system where the user can decide to wait, avoiding an abort, or to abort immediately without waiting. In most cases, neither of these possibilities is offered. Sometimes one possibility is offered, thus improving the perceived quality of the system. A hard-mounted resource in the

NFS file system offers the first possibility. The Stop button in a Web browser offers the second possibility.

This leads to a principle of containment: "abnormal" behavior of any layer should be containable by a higher layer. Therefore the abnormal layer should not make any irrevocable decisions, such as aborting execution, unless this is desired by the programmer. The programmer decides which higher layer is competent to handle the problem. The higher layer should be able to take any reasonable course of action. In our example, this means that time-out should not be a wired-in property of a system. The programmer should decide whether or not to have a time-out, and what to do after a given time has elapsed (one of the possibilities being to continue waiting).

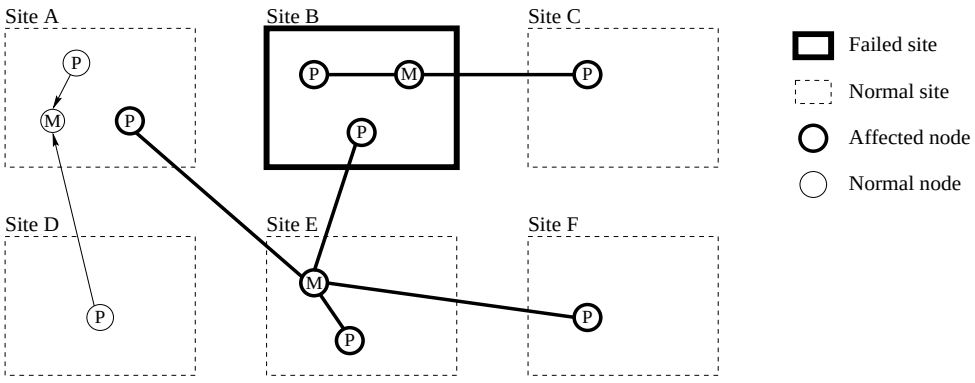


Figure 6.24: Remote detection of site failure in Distributed Oz

6.7.2 Failures in the distribution graph

The external cause of a failure in Distributed Oz is the failure of one or more sites or of part of the network. This shows up in the distribution graph at the level of access structures. We say an access structure is *affected* if it has at least one node on a failed site or if it has at least one link across a failed network. An affected access structure can in many cases continue to work normally, e.g., an object can still be used even if it has a remote reference on a failed site. An access structure is *failed* if normal sites can no longer do operations on it. This happens if a crucial part of the access structure, e.g., the manager node, is inaccessible because it is on a failed site or across a failed network. Figure 6.24 shows a system that covers six sites. Site B has failed; sites A, C, E, and F have affected nodes; and site D has only normal nodes. We assume that sites are designed to satisfy the fail-stop property, i.e., site failures happen instantly and are permanent. Networks may have temporary failures, i.e., the network may return to normal. An access structure can therefore have both temporary and permanent failures.

6.7.3 Handlers and watchers

Distributed Oz detects failure at the level of access structures, which shows up in the language as single language entities, e.g., objects, variables, and ports. The default behavior is that an attempted operation on an entity blocks indefinitely if there is a problem in doing the operation. Any other behavior must be specified explicitly by the programmer. We propose to do this by installing handlers and watchers on the entity. A *handler* is invoked if an error is detected when trying to do an operation on the entity (lazy detection). A *watcher* is invoked when an error is detected for an entity, even if no operation is attempted on the entity (eager detection).

The semantics of handlers and watchers is simple. If an operation is attempted on a failed entity, then the operation is replaced by a call of the handler, if one exists with a valid trigger condition. If the system discovers that an entity has failed, then every watcher with a matching trigger condition is immediately made runnable in a newly-created thread. Handlers and watchers have two arguments, namely the failed entity itself and information about the type of error.

Handlers may be installed on entities per site and per thread. Per site and per entity, there is at most one site-based handler and at most one thread-based handler per thread. Thread-based handlers override site-based handlers, i.e., where both apply only the thread-based handler is invoked. Watchers may be installed on entities per site. There may be any number of watchers per entity on a given site.

Handlers and watchers are installed by builtins with the following three arguments: the entity, control information, and the handler or watcher itself, which is a two-argument procedure. The control information gives the type of error for which the handler or watcher should be invoked. In the case of handlers, the control information also stipulates whether the handler is installed on a site or thread basis, and whether after handler invocation the operation should be retried.

6.7.4 Classifying possible failures

Failure detection distinguishes between four classes of failure. A failure can be either temporary or permanent. These are further subdivided into home and foreign failures. Home failures prevent the current site from performing operations on the entity, while foreign failures indicate that there is a problem among other sites sharing the entity preventing some or all of them from performing operations on the entity. Handlers are triggered on home failures. Watchers may be triggered on home as well as foreign failures. Foreign failures give the site an indication that there is more than network latency behind a lack of activity by other sites.

6.7.5 Distributed garbage collection with failures

If a site fails, then credit is lost for all affected access structures whose manager is still working. These access structures will not be reclaimed unless we introduce another idea. A technique that is successfully being used in existing systems, e.g., Java

RMI [90], is a *lease-based* reference-counting mechanism. This technique can also be used together with the credit mechanism. Any site that has credit for an access structure must periodically renew its lease by sending a message to the manager. If the manager does not receive at least one renewal message within a given time period, then the manager can be reclaimed.

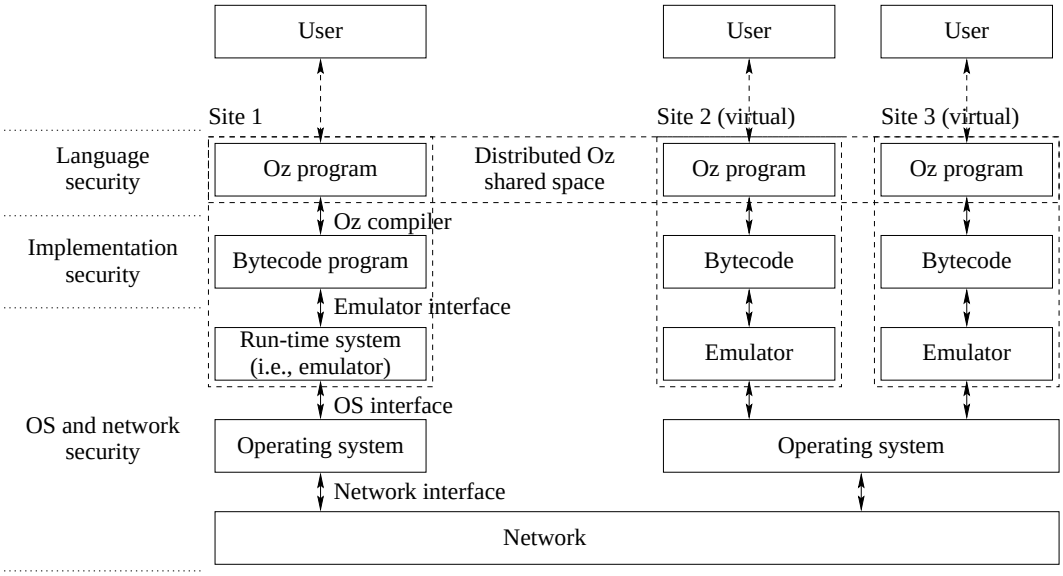


Figure 6.25: Security issues in Distributed Oz

6.8 Resource control and security

An application is *secure* if it can continue to fulfill its specification despite intentional (i.e., malicious) failures of its components. Resource control and security are global issues, i.e., they cross abstraction boundaries [8], just like fault tolerance. The issues must therefore be addressed at each layer. We briefly discuss what can be done in Distributed Oz. Fault tolerance and security have much in common [78], including the reliance on containment and redundancy. But they focus on very different classes of failures. For example, a crucial part of security is resource control because exhausting resources is a common technique to provoke intentional failures ("denial of service" attacks). Although important, resource control is less critical for fault tolerance.

Resources are conveniently divided into site and network resources. Site resources include computational resources (memory/processor) and other resources such as file systems and peripherals. The same site resources normally appear in some form at each site layer, i.e., Oz program, emulator, and operating system. In a similar way, security issues appear at each layer (see Figure 6.25):

```
% Create new module ROpen that looks like the standard Open
% but allows only reading and only in the given directory Dir:
proc {NewReadInDir Dir ROpen}
   class ROpenF
      attr fd
      meth init(name:FN)
         % Should verify absence of '..' and '/' in FN!
         fd <- {New Open.file init(name:Dir#FN)}
      end
      meth read(list:CL)
         {@fd read(list:CL)}
      end
   end
in
   ROpen=open(file:ROpenF)
end

% Give limited rights to the untrusted object UntrustedObj:
SandboxOpen = {NewReadInDir "/usr/home/untrusted_foreign/"}
{UntrustedObj setopenmodule(SandboxOpen)}
```

Figure 6.26: Capabilities in Oz

- **Language security** is a property of the language. It guarantees that computations and data are protected from adversaries that stay within the language.

- **Implementation security** is a property of the language implementation in the process. It protects computations and data from adversaries who attempt to interfere with compiled programs, i.e., with the Oz bytecode.

- **Operating system and network security** are properties of the operating system and network. They protect computations and data from adversaries who attempt to interfere with the internals of the Oz emulator and run-time system within an operating system process, and who attempt to interfere with the operating system and the network. Network security is available through secure TCP/IP.

6.8.1 Language security

We provide language security by giving the programmer the means to restrict access to data. Data are represented as references to entities in an abstract shared computation space. The space is *abstract* because it provides a well-defined set of basic operations.

In particular, unrestricted access to memory is forbidden.[15] One can only access data to which one has been given an explicit reference.

A reference to a procedure or an object behaves as a capability. Because of lexical scoping and first-class procedures [4], it is possible to create new capabilities that encapsulate existing ones, thus possibly limiting their rights. For example, Figure 6.26 shows how to give an object limited rights to a file system. Calling `{NewReadInDir Dir ROpen}` creates a new module `ROpen`, which behaves exactly like the system module `Open`, except that it only allows to read files and only in the given directory `Dir`.

Capabilities do not solve all problems in security [134]. They have inherent weaknesses. First, the authorization to do something is given very early, namely when the capability is given and not when the operation is attempted. Second, a capability can be forwarded to anyone and it will continue working. Therefore, a capability-based mechanism needs to be extended–for example with access control based on the identity of the capability's current possessor.

6.8.2 Implementation security

Two important issues in implementation security are site integrity and resource control. These issues appear when code is directly or indirectly passed between sites. For example, sending a procedure to a compute server to execute it (direct) or invoking a method of a mobile object (indirect). The foreign code is not necessarily trustworthy. The importing site should be protected from being corrupted by malicious foreign code, i.e., by invalid Oz bytecode. This is very difficult in general. Typical techniques are bytecode verification and authenticating compilers [125].

The foreign code should be limited in its ability to use the site's computational resources. Monopolizing the processor may starve the site's other concurrent activities. Excessive memory use may exhaust the site's memory, which being extremely difficult to recover from, would effectively crash the site.

The foreign code should be given controlled access to other site resources. Obviously, untrusted code cannot be given unlimited access to site-specific resources such as file systems. It is possible, but not practical, to forbid access to all site-specific resources (just as it is not practical to forbid all access to basic computational resources!). Better is to provide limited capabilities.

6.8.3 Virtual sites

To partially provide implementation security in Distributed Oz, we propose the mechanism of *virtual sites* (see Figure 6.25). A site can spawn slave virtual sites, which behave exactly like standard sites except that the master monitors and controls the slaves. If the slave crashes then the master is notified but not otherwise affected. The master controls slaves' resources, including their computational resources and other

[15] For example, both examining data representations (type casts) and calculating addresses (pointer arithmetic) are forbidden.

resources such as access to file systems. For example, the slave site might be given the possibility to create and delete files in one specific directory but nowhere else.

Within the limitations imposed by the master, a virtual site behaves almost exactly the same as an ordinary site. It may share Oz entities with the master site or any other site. The difference is that the virtual site shares the same machine. Communication is more efficient since there is no network layer. To take advantage of the protection and resource control mechanisms of the operating system, a slave site will normally live in a different process than its master.

Virtual sites can be used to exploit the computational resources of shared-memory multiprocessors. Simply allocate one virtual site per processor. Because communication overheads are lower, this is more efficient than parallelism over the network. Whether the parallelism leads to an effective speedup of course still depends on these overheads.

6.9 Conclusion

Distributed programming is of major importance today, yet it remains difficult. We present a design for a distributed programming language, Distributed Oz, that fully separates an application's functionality from its distribution structure. Distributed Oz is a conservative extension to the existing centralized Oz language. Oz is a concurrent object-oriented language that is state-aware and that has dataflow synchronization. Oz programs can be ported almost immediately to Distributed Oz, which is implemented and publicly available. We are experimenting with distributed applications including collaborative tools, compute servers, and techniques for using centralized applications in distributed settings [18].

Distributed Oz is very much work in progress. We present preliminary designs that conservatively extend the language with models for open computing, fault tolerance, and resource control. These designs are being implemented and extended.

Acknowledgements

We thank the numerous people at SICS, DFKI, and UCL that have contributed to Distributed Oz and the referees for useful comments. We thank Donatien Grolaux for suggesting transaction objects. Seif Haridi and Per Brand are supported by the Swedish national board for industrial and technical development (NUTEK) and SICS. Christian Schulte is supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FKZ ITW 9601) and the Esprit Working Group CCL-II (EP 22457).

Chapter 7

Mobile Objects in Distributed Oz

Mobile Objects in Distributed Oz

PETER VAN ROY
Université Catholique de Louvain

SEIF HARIDI and PER BRAND
Swedish Institute of Computer Science
and

GERT SMOLKA, MICHAEL MEHL, and RALF SCHEIDHAUER
German Research Center For Artifi cial Intelligence (DFKI)

7.1 Abstract

Some of the most difficult questions to answer when designing a distributed application are related to mobility: what information to transfer between sites and when and how to transfer it. Network-transparent distribution, the property that a program's behavior is independent of how it is partitioned among sites, does not directly address these questions. Therefore we propose to extend all language entities with a network behavior that enables *efficient* distributed programming by giving the programmer a simple and predictable control over network communication patterns. In particular, we show how to give objects an arbitrary mobility behavior that is independent of the object's definition. In this way, the syntax and semantics of objects are the same regardless of whether they are used as stationary servers, mobile agents, or simply as caches. These ideas have been implemented in Distributed Oz, a concurrent object-oriented language that is state aware and has dataflow synchronization. We prove that the implementation of objects in Distributed Oz is network transparent. To satisfy the predictability condition, the implementation avoids forwarding chains through intermediate sites. The implementation is an extension to the publicly available DFKI Oz 2.0 system.

7.2 Introduction

The distinguishing feature of a distributed computation is that it is partitioned among sites. It is therefore important to be able to easily and efficiently transfer both computations and information between sites. Yet, when the number of sites increases without bounds, the programmer must not be burdened with writing separate programs for each site and explicitly managing the communications between them. We conclude that there are two conflicting goals in designing a language for distributed programming. First, the language should be network transparent, i.e., computations behave correctly independently of how they are partitioned between sites. Second, the language should give simple and predictable control over network communication patterns. The main contribution of this article is to present a language, Distributed Oz, that satisfies these two goals. The design has two key ideas: first, to define the language in terms of *two* semantics, a language semantics and a distributed semantics that refines it to take network behavior into account, and second, to incorporate mobility in a fundamental way in the distributed semantics.

7.2.1 Object Mobility

Making mobility a primitive concept makes it possible to define efficient networked objects whose mobility can be precisely controlled (see Section 7.6.4). The object can change sites on its own or on request. The object does not leave a trail, i.e., it does not leave behind aliases or surrogate objects to forward messages when it changes sites. There is no "proxy explosion" problem when an object is passed repeatedly between two sites [42]. Many sites may send messages to the object. It is eventually true

that messages sent will go to the object in a single network hop, no matter how many times the object moves. There is no difference in syntax or computational behavior between these objects and stationary objects. No published system has objects with these abilities. In particular, Emerald [72], Obliq [21], and Java with Remote Method Invocation [120] all suffer from the aliasing problem to some extent. One of the contributions of this article is to show how to provide mobile objects with predictable network behavior, i.e., without aliasing, in a simple and straightforward way.

7.2.2 Two Semantics

The basic design principle of Distributed Oz is to distinguish clearly between the *language* semantics and the *distributed* semantics. Distributed Oz has the same language semantics as Oz 2, a concurrent object-oriented language that is state aware and has dataflow synchronization. The object system has a simple formal foundation and yet contains all the features required in a modern concurrent language. Detailed information about the language and its object system can be found in [51] and [60].[1] Implementations of Oz and its successor Oz 2 have been used in many research projects [[13]; [22]; [39]; [40]; [63]; [61]; [110]; [135]]. To be self-contained, this article uses a subset of Oz 2 syntax that directly corresponds to its semantics.

The distributed semantics extends the language semantics to take into account the notion of site. It defines the network operations invoked when a computation is partitioned on multiple sites. There is no distribution layer added on top of an existing centralized system. Rather, all language entities are given a network behavior that respects the same language semantics they have when executing locally. By a *language entity* we mean a basic data item of the language, such as an object, a procedure, a thread, or a record. Figure 7.3 classifies the entities and summarizes their distributed semantics. Network operations[2] are predictable, which gives the programmer the ability to manage network communications.

7.2.3 Developing an Application

Developing an application is separated into two independent parts. First, the application is written without explicitly partitioning the computation among sites. One can in fact check the *safety* and *liveness* properties[3] of the application by running it on one site. Second, the application is made *efficient* by controlling the mobility of its entities. For example, some objects may be placed on certain sites, and other objects may be given a particular mobile behavior. The shared graphic editor of Section 7.3 is designed according to this approach.

[1] See also `http://www.ps.uni-sb.de`.

[2] In terms of the number of network hops.

[3] A fortiori, correctness and termination for nonreactive applications.

7.2.4 Mobility Control and State

The distributed semantics extends the language semantics with *mobility control*. In general terms, mobility control is the ability for stateful entities to migrate between sites or to remain stationary at one site, according to the programmer's intention [56]. The programmer can use mobility control to program the desired network communication patterns in a straightforward way. For example, to reduce network latency a mobile object can behave as a state cache. This is illustrated by the shared graphic editor of Section 7.3.

By *stateful entities* we mean entities that change over time, i.e., they are defined by a sequence of states, where a *state* can be any entity. At any given moment, a stateful entity is localized to a particular site, called its *home* site. Stateful entities are of two kinds, called *cells* and *ports*, that are respectively mobile and stationary. Objects are defined in terms of cells. The mobility behavior of an object is defined in terms of cells and ports. The language semantics of these entities is given in Section 7.5, and their distributed semantics is given in Section 7.6. The implementation contains a mobile state protocol that implements the language semantics of cells while allowing the cell's state[4] to efficiently migrate between sites.

It is important that *all* language entities have a well-defined network behavior. For example, the language provides network references to procedures. A procedure is stateless, i.e., its definition does not change over time. Calling the procedure locally or remotely gives the same results. Disregarding sites, it behaves identically to a centralized procedure application. Passing a procedure to a remote site causes a network reference to be created to the procedure. Calling the procedure causes it to be replicated to the calling site.[5] The procedure's external references will be either replicated or remotely referenced, depending on what kind of entity they are. Only the first call will have a network overhead if the procedure does not yet exist on the site.

7.2.5 Overview of the Article

This article consists of eight parts and an appendix. Section 7.3 presents an example application, a shared graphic editor, that illustrates one way to use mobile objects to reduce network latency. Section 7.4 lays the foundation for our design by reasoning from four general requirements for distributed programming. Section 7.5 summarizes the language semantics of Distributed Oz in terms of these requirements. Section 7.6 defines and justifies the distributed semantics of Distributed Oz. We show by example how easy it is to code various kinds of migratory behavior using cells and ports. Section 7.7 outlines how mobility is introduced into the language semantics and specifies a mobile state protocol for cells. Section 7.8 summarizes the system architecture and situates the protocol in it. Section 7.9 compares the present design with distributed shared memory, Emerald, and Obliq. Section 7.10 summarizes the main contributions

[4] More precisely, its *content-edge*, which is defined in Section 7.5.1.

[5] This is not the same as an RPC. To get the effect of an RPC, a stationary entity (such as a port) must be introduced.

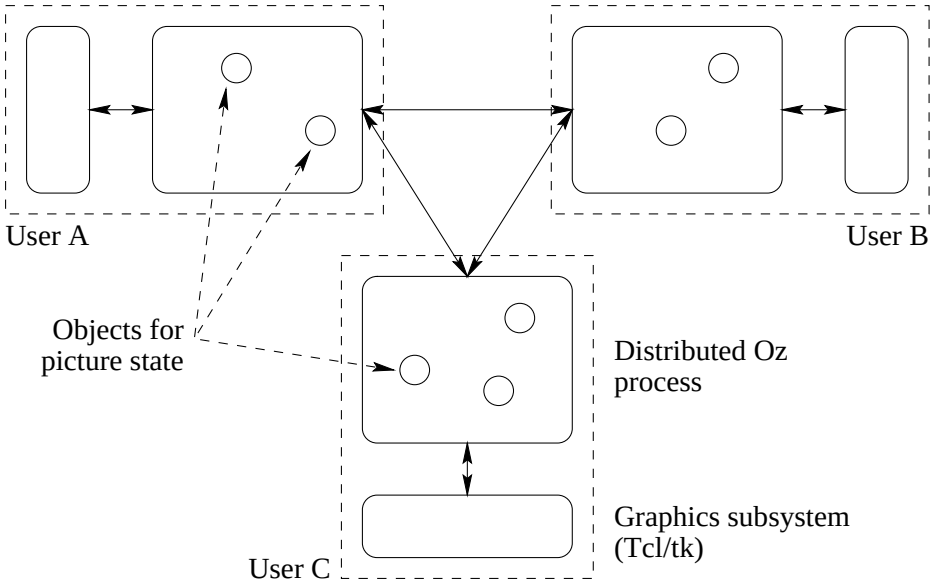


Figure 7.1: A shared graphic editor.

and the status of the project. Finally, Appendix 7.11.1 gives a formal proof that the mobile state protocol implements the language semantics for cells.

7.3 A Shared Graphic Editor

Writing an efficient distributed application can be much simplified by using network transparency and mobility. We have substantiated this claim by designing and implementing a prototype shared graphic editor, an application which is useful in a collaborative work environment. The editor is seen by an arbitrary number of users. We wish the editor to behave like a shared virtual environment. This implies the following set of requirements. We require that all users be able to make updates to the drawing at any time, that each user sees his or her own updates without any noticeable delays, and that the updates must be visible to all users in real time. Furthermore, we require that the same graphical entity can be updated by multiple users. This is useful in a collaborative CAD environment when editing complex graphic designs. Finally, we require that all updates are sequentially consistent, i.e., each user has exactly the same view of the drawing. The last two requirements are what makes the application interesting. Using multicast to update each user's visual representation, as is done for example in the LBL Whiteboard application, [6] does not satisfy the last two requirements.

Figure 7.1 outlines the architecture of our prototype. The drawing state is represented as a set of objects. These objects denote graphical entities such as geometric shapes and freehand drawing pads. When a user updates the drawing, either a new object is created or a message is sent to modify the state of an existing object. The object

[6] Available at `http://mice.ed.ac.uk/mice/archive`.

then posts the update to a distributed agenda. The agenda sends the update to all users so they can update their displays. The users see a shared stream, which guarantees sequential consistency.

New users can connect themselves to the editor at any time using the open computing ability of Distributed Oz. The mechanism is extremely simple: the implementation provides primitives for saving and loading a language entity in a file named by a URL. A URL is an Ascii string that names a globally unique file and is recognized by HTTP clients and servers. We use a URL because it provides us with a convenient global address space. The graphic editor saves to a file a reference to the object that is responsible for managing new users. By loading the file, a new user gets a reference to the object. The two computations then reference the same object. This transparently opens a connection between two sites in the two computations. From that point onward, the computation space is shared. When there are no more references between two sites in a computation, then the connection between them is closed by the garbage collector. Computations can therefore connect and disconnect at will. The issue of how to manage the shared names represented by the URLs leads us into the area of multiagent computations. This is beyond the scope of the article, however.

The design was initially built with stationary objects only. This satisfies all requirements except performance. It works well on low-latency networks such as LANs, but performance is poor when users who are far apart, e.g., in Sweden, Belgium, and Germany, try to draw freehand sketches or any other graphical entity that needs continuous feedback. This is because a freehand sketch consists of many small line segments being drawn in a short time. In our implementation, up to 30 motion events per second are sent from the graphics subsystem to the Oz process. Each line segment requires updating the drawing pad state and sending this update to all users. If the state is remote, then the latency for one update is often several hundred milliseconds or more, with a large variance.

To solve the latency problem, we refine the design to represent the picture state and the distributed agenda as freely mobile objects rather than stationary objects. The effect of this refinement is that parts of the picture state are cached at sites that modify them. Implementing the refinement requires changing some of the calls that create new objects. In all, less than 10 lines of code out of 500 have to be changed. With these changes, freehand sketches do not need any network operations to update the local display, so performance is satisfactory. Remote users see the sketch being made in real time, with a delay equal to the network latency.

This illustrates the two-part approach for building applications in Distributed Oz. First, build and test the application using stationary objects. Second, reduce latency by carefully selecting a few objects and changing their mobility behavior. Because of transparency, this can be done with quite minor changes to the code of the application itself. In both the stationary and mobile designs, fault tolerance is a separate issue that must be taken into account explicitly. It can be done by recording on a reliable site a log of all display events. Crashed users disappear, and new users are sent a compressed version of the log.

In general, mobile objects are useful both for fine-grain mobility (caching of object

Requirements	Mechanisms
Network transparency	Shared computation space, concurrency
Flexible network awareness	State awareness, mobility control
Latency tolerance	Concurrency, caching, dataflow synchronization, asynchronous ordered communication
Language security	Capability-based computation space, lexical scoping, first-class procedures

Table 7.1: System Requirements and Some of their Mechanisms

state) as well as coarse-grain mobility (explicit transfer of groups of objects). The key ability that the system must provide is transparent control of mobility, i.e., control that is independent of the object's functionality. Section 7.6.4 shows how this is done in Distributed Oz.

7.4 Language Properties

In order to provide a firm base for the language design, we start from four requirements that are generally agreed to be important in a distributed setting. We then propose a set of mechanisms that are sufficient to satisfy these requirements. The four requirements are network transparency, flexible network awareness, latency tolerance, and language security. Table 7.1 summarizes the requirements and their enabling mechanisms. Section 7.5 presents a design that contains all the mechanisms. For brevity, we give only a summary of the fault model. Other important requirements such as resource management and network security will be presented elsewhere.

It is not obvious that the four requirements can be satisfied simultaneously. In particular, achieving both network transparency and flexible network awareness may seem inherently impossible. It becomes possible by carefully distinguishing between the language semantics and distributed semantics.

7.4.1 Network Transparency

Network transparency means that computations behave in the same way independent of the distribution structure.[7] That is, the language semantics is obeyed independent of how the computation is partitioned onto multiple sites. This requires a *distributed shared computation space*, which provides the illusion of a single networkwide address space for all entities (including threads, objects, and procedures). The distinction

[7] The terms "network transparency" and "network awareness" were first introduced by Cardelli [21].

between local references (on the same site) and remote references (to another site) is invisible to the programmer. Consistency of remote access is explained in Section 7.6. For reasons of security, the computation space is not just a shared memory. This is explained below.

To be practical, a network-transparent system must be able to express all important distributed-programming idioms. Many of these do parallel execution, e.g., multiple clients accessing multiple servers. Therefore the language must be *concurrent*, i.e., allow for multiple computational activities (called "threads") that coexist and evolve independently.

7.4.2 Flexible Network Awareness

Flexible network awareness means two things: predictability and programmability. Network communication patterns should be simply and predictably derived from the language entities. In addition, the communication patterns provided should be flexible enough to program the desired network behavior. The resulting distributed semantics gives the programmer explicit control over network communication patterns.

The basic insight to achieve flexible network awareness is that for efficiency, stateful data (e.g., objects) must at any instant reside on exactly one site (the home site).[8] On the other hand, stateless data (e.g., procedures or values) can safely be replicated, i.e., copied to another site. It is therefore useful for the language to distinguish between these two kinds of data, that is, it is *state aware*. Replication is used in first instance to improve the network behavior of stateless data.

Mobility control is the ability of a home site to change (mobility) or to remain the same (stationarity). With the concepts of state awareness and mobility control, the programmer can express any desired network communication pattern. In the design described here, entities have three basic network behaviors. Mobile entities migrate to each remote site invoking them. The implementation is careful to make the network behavior of mobile entities predictable by using the appropriate distributed algorithm. Stationary entities require a network operation on each remote invocation. Replicable entities are copied to each remote site requesting the entity. More complex network behaviors are built from these three.

7.4.3 Latency Tolerance

Latency tolerance means that the efficiency of computations is affected as little as possible by network delay. Distributed Oz provides four basic mechanisms for latency tolerance. *Concurrency* provides latency tolerance between threads: while one thread waits for the network, other threads can continue.

Caching improves latency tolerance by increasing the locality of computations. Caching stateless entities amounts to replicating them. Caching stateful entities requires a coherence protocol. The mobile state protocol of Distributed Oz guarantees

[8] They can of course be referenced from any site.

coherence and is optimized for frequent state updates and moves. The current design does not yet have a replacement policy, i.e., there are no special provisions for resource management.

Dataflow synchronization allows computations to stall only on data dependency (not on send or receive). Well-known techniques to achieve this are *futures* [49], *I-structures* [64], and *logic variables* [15, 114]. We use logic variables because they have great expressive power, are easily implemented efficiently, and are consistent with the semantics of a state-aware language (see Section 7.5.1). *Asynchronous ordered communication* generalizes logic variables by adding a form of buffering. This is provided by *ports* (see Section 7.5.2).

7.4.4 Language Security

Language security means that the language guarantees computations and data are protected from adversaries that use only the language. It is important to distinguish between *language* security and *implementation* security. Implementation security means that computations and data are protected from adversaries that have access to the system's implementation. This is beyond the scope of the article, though. We provide language security by giving the programmer the means to restrict access to data. Data are represented as references to entities in an abstract shared computation space. The space is *abstract* because it provides a well-defined set of basic operations. In particular, unrestricted access to memory is forbidden.[9] One can only access data to which one has been given an explicit reference. This is controlled through lexical scoping and first-class procedures [4]. *Lexical scoping* implies that a procedure's initial set of external references is determined by its static structure. Other references can be passed around explicitly during execution. Having *first-class procedures* implies that procedures can be created and applied dynamically and that references to them can be passed around:

```
local P in  % Declare P in a large scope
   % At site 1: Declare X in a limited scope
   local X in
      % Define procedure P
      proc {P ...} ... end   % X is visible inside P
   end

   % At site 2:
   local Q in
      % Define procedure Q
      proc {Q ...} ... end   % X is not visible inside Q
   end
end
```

[9] For example, both examining data representations (type casts) and calculating addresses (pointer arithmetic) are forbidden.

S ::=	$S\ S$	Sequence
|	X=f(l_1:Y_1 ... l_n:Y_n) |	Value
	X=<number> | X=<atom> | {NewName X}	
|	**local** X_1 ... X_n **in** S **end** | X=Y	Variable
|	**proc** {X Y_1 ... Y_n} S **end** | {X Y_1 ... Y_n}	Procedure
|	{NewCell Y X} | {Exchange X Y Z} | {Access X Y}	State
|	**if** X=Y **then** S **else** S **end**	Conditional
|	**thread** S **end** | {GetThreadId X}	Thread
|	**try** S **catch** X **then** S **end** | **raise** X **end**	Exception

Table 7.2: Summary of OPM Syntax

Procedure P can access X, but procedure Q cannot. However, since Q can access P, this gives Q indirect access to X. Therefore Q has some rights to X, namely those that it has through P. Passing procedures around thus transfers access rights, which gives the effect of capabilities.

7.5 Language Semantics

Distributed Oz, i.e., Oz 2, is a simple language that satisfies all the requirements of the previous section. Distributed Oz is dynamically typed, i.e., its type structure is checked at run-time. This simplifies programming in an open distributed environment. The language is fully compositional, i.e., all language constructs that contain a statement may be arbitrarily nested. All examples given below work in both centralized and distributed settings.

Distributed Oz is defined by transforming all its statements into statements of a small kernel language, called OPM (Oz Programming Model) [115, 116]. OPM is a concurrent programming model with an interleaving semantics. It has three innovative features. First, it uses dataflow synchronization through logic variables as a basic mechanism of control. Second, it makes an explicit distinction between stateless references (logic variables) and stateful references (cells). Finally, it is a unified model that subsumes higher-order functional and object-oriented programming.

The basic entities of OPM are values, logic variables, procedures, cells, and threads. A *value* is unchanging and is the most primitive data item that the language semantics recognizes. For all entities but logic variables, an *entity* is a group of one or more values that are useful from the programmer's point of view. A record entity consists of one value (the record itself), and a cell entity consists of two values (its name and content). The full language provides syntactic support for additional entities including objects, classes, and ports. The system hides their efficient implementation while respecting their definitions. All entities except for logic variables have one value that is used to identify and reference them.

7.5.1 Oz Programming Model

This section summarizes OPM, the formal model underlying Oz 2. Readers interested mainly in the object system may skim directly to Section 7.5.2 on first reading. A program written in OPM consists of a (compound) statement containing value descriptions, variable declarations, procedure definitions and calls, state declarations and updates, conditionals, thread declarations, and exception handling. This is summarized in Table 7.2.

Computation takes place in a *computation space* hosting a number of sequential *threads* connected to a single shared *store*. The store contains three compartments: a set of variables, each with its binding if bound, a set of procedure definitions, and a set of cells. Variable bindings and procedure definitions are immutable. Cells are updatable, as explained below. Each thread consists of a sequence of statements. Computation proceeds by *reduction of statements* that interact with the store and may create new threads. Reduction is fair between threads. Once a statement becomes reducible, it stays reducible.

Value Description

The values provided are records (including lists), numbers, literals (names and atoms), and closures. Except for names and closures, these values are defined in the usual way. Closures are created as part of procedures and are only accessible through the procedure name. The other values can be written explicitly or referred to by variables:

```
local V W X Y Z H T in
   V=queue(head:H tail:T)  % Record
   W=H|T                   % Record (representing a list)
   X=333667                % Number
   Y=foo                   % Literal (atom)
   {NewName Z}             % Literal (name)
end
```

A *name* has no external representation and hence cannot be forged within the language. The call `{NewName Z}` creates a new name that is unique systemwide. Names are used to identify cells, procedures, and threads. Names can be used to add hidden functionality to entities. For example, the server loop of Section 7.6.4 is stopped with a secret name. Names are to language security what capabilities are to implementation security.

Variable Declaration

All variables are logic variables. They must be declared in an explicit scope bracketed by **`local`** and **`end`**. The system enforces that a variable always refers to the same value. A variable starts out with its value unknown. The value becomes known by

binding the variable, i.e., after executing the binding `X=V`, variable `X` has value `V`.[10] The binding operation is called *incremental tell* [116]. In essence, incremental tell only adds variable bindings that are consistent with existing bindings in the store.

Any attempt to use a variable's value will block the thread making the attempt until the value is known. From the viewpoint of the thread, this is unobservable. It affects only the relative execution rate of the thread with respect to other threads. Therefore logic variables introduce a fundamental dataflow element in the execution of OPM.

Binding variables is the basic mechanism of communication and synchronization in OPM. It decouples the acts of *sending* and *receiving* a value from the acts of *calculating* and *using* that value. A logic variable can be passed to a user thread before it is bound:

```
local X in                  % Declare X, value is unknown
   thread {Consumer X} end % Create thread, use X
   {Producer X}             % Calculate value of X
end
```

The call `{Consumer X}` can start executing immediately. Its thread blocks only if `X`'s value is not available at the moment it is needed. We assume that the call `{Producer X}` will eventually calculate `X`'s value.

Procedure Definition and Call

Both procedure definitions and calls are executed at run-time. For example, the following code defines `MakeAdder`, which itself defines `Add3`:

```
local
   MakeAdder Add3 X Y
in
   proc {MakeAdder N AddN}  % Procedure definition
      proc {AddN X Y} Y=X+N end
   end
   {MakeAdder 3 Add3}       % Procedure call
   {Add3 10 X}   % X gets the value 13
   {Add3 1 Y}    % Y gets the value 4
end
```

Executing the call `{MakeAdder 3 Add3}` defines `Add3`, a two-argument procedure that adds 3 to its first argument. Executing a procedure definition creates a pair of a *name* and a *closure*. A variable referring to a procedure actually refers to the name. When calling the procedure, the name is recognized as corresponding to a procedure definition. A closure is a value that contains the procedure code and the external references of the procedure (which are given by lexical scoping).

[10] Variables may be bound to other variables. An exception is raised if there is an attempt to bind a variable to two different values.

State Declaration and Update

Variables always refer to values, which never change. *Stateful* data must be declared explicitly during execution by creating a *cell*. The call `{NewCell X C}` creates a pair of a new name (referred to by `C`) and an initial content `X`. The content can be any value. The pair is called the *content-edge* of the cell. Two other operations on cells are an atomic read-and-write (exchange) and an atomic read (access). The call `{Exchange C X Y}` atomically updates the cell with a new content `Y` and invokes the binding of `X` to the old content, and the call `{Access C X}` invokes the binding of `X` to the cell content.

```
local C X1 X2 X3
in
   {NewCell bing C}     % C's cell has initial content bing
   {Exchange C X1 bang}
   % X1 bound to bing; new content is bang
   {Exchange C X2 bong(me:C was:X2)}
   % X2 bound to bang; new content is bong(me:C was:X2)
   {Access C X3}
   % X3 bound to bong(me:C was:X2)
end
```

Cells and threads are the only stateful entities in OPM. The other stateful entities in Distributed Oz, namely objects and ports, are defined in terms of cells.

Conditional

There is a single conditional statement. The condition must be a test on the values of data structures:

```
local X in
   thread  % Put conditional in its own thread
      % Block until can decide the condition
      if X=yes then Z=no else Z=yes end
   end
   X=no    % Now decide the condition: it is false
end
```

The conditional blocks its thread until it has enough information about the value of `X` to decide whether `X=yes` is true or false. In this case, the binding `X=no` makes it false. Local logic variables may be introduced in the condition. Their scope extends to the end of the **`then`** branch.

Thread Declaration

Execution consists of the preemptive and fair reduction of a set of threads. Each thread executes in strictly sequential manner. A thread will *block*, or suspend execution, if a value it needs is not available. The thread becomes reducible again when the value becomes available. Concurrency is introduced explicitly by creating a new thread:

```
% Define a procedure and run it in a new thread
local Loop in
   proc {Loop N} {Loop N+1} end
   thread {Loop 0} end
end
```

Each thread is identified uniquely by a name, which can be obtained by executing `{GetThreadID T}` in the thread. With the thread name, it is possible to send commands to the thread, e.g., suspend, resume, and set priority. Thread names can be compared to test whether two threads are the same thread.

Exception Handling

Exception handling is an extension to a thread's strictly sequential control flow that allows to jump out from within a given scope. For example, `AlwaysCalcX` will return a value for `X` in cases when `CalcX` cannot:

```
proc {AlwaysCalcX CalcX A X}
   try
      local Z in
         {CalcX A Z}
         Z=X  % Bind X if there is no exception in CalcX
      end
   catch E then
      {FailFix A X}
   end
end
```

The **`try`** S_1 **`catch`** E **`then`** S_2 **`end`** defines a context for exception handling in the current thread. If an exception `T` is raised during the execution of S_1 in the same thread then control will transfer to the **`catch`** clause of the innermost **`try`**. If `T` matches E then S_2 is executed, and the **`try`** is exited; otherwise the exception is reraised at an outer level. User-defined exceptions can be raised by the statement **`raise`** `T` **`end`**.

7.5.2 Compound Entities

Distributed Oz provides the following two additional derived entities over OPM:

- Concurrent objects with explicit reentrant locking. There is syntactic support for classes with multiple inheritance and late binding.

- Ports, which are asynchronous channels. Ports are related to *M-structures* [16].

These entities are entirely defined in terms of OPM. They are provided because they are useful abstractions. In Section 7.6 they are given a specific distributed semantics.

Concurrent Objects

An object in Oz 2 is defined in OPM as a one-argument procedure. The procedure references a cell which is used to hold the object's internal state. State update and access are done with cell exchange and access. The procedure's argument is the message, which indexes into the method table. Methods are procedures that are passed the message and the object's state. Mutual exclusion of method bodies is supported through explicit reentrant locking.

Class definitions and object declarations are both executed at run-time. A class definition builds the method table and resolves conflicts of multiple inheritance. Like OPM, both class definitions and object declarations are fully compositional with respect to all language features. For example, arbitrary nesting is allowed between class, object, procedure, and thread declarations. The following presents a definition of objects consistent with Oz 2. For more information see [115], [51], [60], and [62].

An object without locking. The following example in Oz 2 syntax defines the class `Counter`:

```
class Counter    % Define class
   attr val:0    % Attribute declaration with initial value
   meth inc      % Method declaration
      val := @val + 1
   end
   meth get(X)   % Method with one argument
      X = @val
   end
   meth reset
      val := 0
   end
end
```

Objects of this class have a single attribute `val` with initial value `0` and the three methods `inc`, `get`, and `reset`. Attribute access is denoted with `@` and attribute assignment with `:=`.

An object with locking. Instances of this class may be accessed from several concurrent threads. To make sure that the methods are mutually exclusive, Oz 2 uses reentrant locking. This is done by using the construct **`lock`** *S* **`end`** on the part of the methods that require exclusive access. This can be done by specializing the `Counter` class as follows:

```
class LCounter from Counter
   prop locking             % Declare an implicit lock
   meth inc
      lock Counter,inc end  % Call the method of superclass
   end
   meth get(X)
      lock Counter,get(X) end
   end
   meth reset
```

```
         lock Counter,reset end
      end
   end
```

The above class declares an implicit lock and specializes the methods of the superclass `Counter`. The notation `Counter,inc` is a static method call to the methods of the `Counter` class. An instance is created using the procedure `New` as follows:

```
C={New LCounter}  % Create instance
{C inc}           % Send message
{C get(X)}        % Return with X=1
```

With the above class definition, the object `C` behaves in a way equivalent to the following code in OPM:

```
proc {NewCounter ?C}
   State   = s(val:{NewCell 0 $})
   Lock    = {NewLock $}
   Methods = m(inc:Inc get:Get reset:Reset)
   proc {Inc M} V W in
      {Lock proc {$} {Exchange State.val V W} W=V+1 end}
   end
   proc {Get M} X in
      M=get(X) {Lock proc {$} {Access State.val X} end}
   end
   proc {Reset M}
      {Lock proc {$} {Exchange State.val _ 0} end}
   end
in
   proc {C Message}
   M in
      {Label Message M}
      {Methods.M Message}
   end
end
```

This example introduces four syntactic short-cuts which will be used freely from now on. First, the question mark in the argument `?C` is a comment to the programmer that `C` is an output. Second, we omit the **`local`** and **`end`** keywords for new local variables in a procedure or a conditional. Third, all variables occurring before the **`in`** and either as procedure names or to the left-hand side of equalities are newly declared. Fourth, we add a nesting notation for statements, so that `Lock={NewLock $}` is equivalent to `{NewLock Lock}`. The dollar symbol is used as a placeholder. We use the notation **`proc`** `{$ ...}` `...` **`end`** for anonymous procedures.

The procedure `{NewLock Lock}` returns a reentrant lock named `Lock`. In OPM a reentrant lock is a procedure that takes another procedure `P` as argument and executes `P` in a critical section. The lock is thread-reentrant in the sense that it allows the same thread to enter other critical sections protected by the same lock. Other threads trying to acquire the lock will wait until `P` is completed. The definition of `NewLock` in OPM

will be given shortly. As we will see, thread-reentrant locking is modeled in OPM using cells and logic variables.

The procedure `NewCounter` defines the variables `State`, `Lock`, and `Methods`. `State` contains the state of the object defined as a record of cells; `Lock` is the lock; and `Methods` is the method table. Both the state and lock are encapsulated in the object by lexical scoping. The call `{NewCounter C}` returns a procedure `C` representing the object. This procedure, when given a message, will select the appropriate method from the method table and apply the method to the message. The call `{Label M X}` returns record `M`'s label in `X`.

Reentrant locking in OPM. A thread-reentrant lock allows the same thread to reenter the lock, i.e., to enter a dynamically-nested critical region guarded by the same lock. Such a lock can be secured by at most one thread at a time. Concurrent threads that attempt to secure the same lock are queued. When the lock is released, it is granted to the thread standing first in line. Thread-reentrant locks can be modeled by the procedure `NewLock` defined as follows:

```
proc {NewLock ?Lock}
   Token   = {NewCell unit $}
   Current = {NewCell unit $}
in
   proc {Lock Code}
      ThisThread={GetThreadID $}
      LockedThread
   in
      {Access Current LockedThread}
      if ThisThread=LockedThread then
         {Code}
      else Old New in
         {Exchange Token Old New}
         {Wait Old}
         {Exchange Current _ ThisThread}
         try
            {Code}
         finally
            {Exchange Current _ unit}
            New=unit
         end
      end
   end
end
```

This assumes that each thread has a unique identifier `T` that is different from the literal **`unit`** and that is obtained by calling the procedure `{GetThreadID T}`. The `{Wait Old}` call blocks the thread until `Old`'s value is known. The **`try ... finally`** *S* **`end`** is syntactic sugar that ensures *S* is executed in both the normal and exceptional cases, i.e., an exception will not prevent the lock from being released.

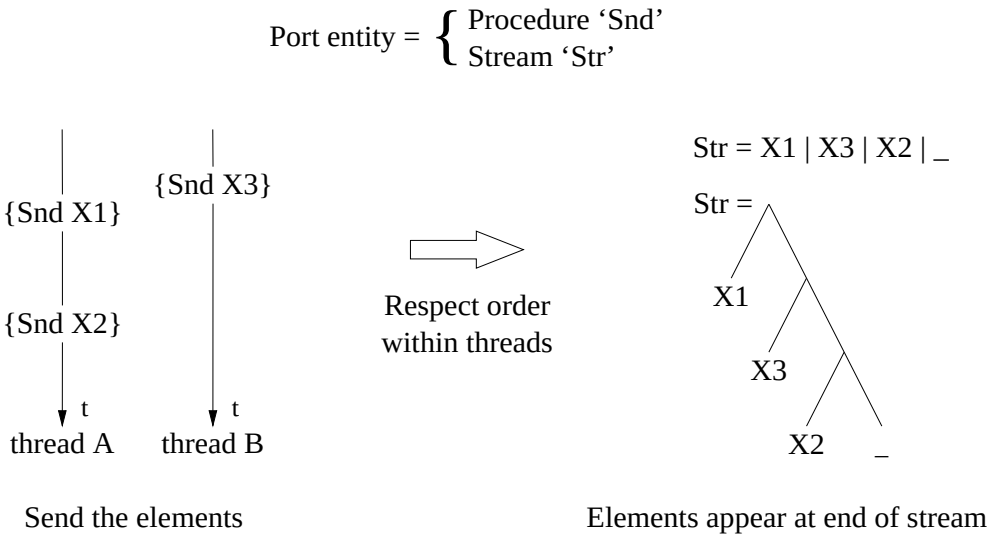


Figure 7.2: Ports: asynchronous channels with multicast ability.

Ports

A *port* is an asynchronous channel that supports ordered many-to-one and many-to-many communication [70]. A port consists of a send procedure and a stream (see Figure 7.2). A *stream* is a list whose tail is a logic variable. Sends are asynchronous and may be invoked concurrently. The entries sent appear at the end of the stream. The send order is maintained between entries that are sent from within the same thread. No guarantees of order are given between threads.

A reader can wait until the stream's tail becomes known. Since the stream is stateless, it supports any number of concurrent readers. Multiple readers waiting on the same tail can be informed of the value simultaneously, thus providing many-to-many communication. Adding an element to the stream binds the stream's tail to pairs (cons cells) containing the entry and a logic variable as the new tail.

Within OPM one can define a port as a send procedure and a list. The procedure refers to a cell which holds the current tail of the list. Ports are created by `NewPort`, which can be defined as follows:[11]

```
proc {NewPort Str ?Snd}
   C={NewCell Str $}
in
   proc {Snd Message}
   Old New in
      {Exchange C Old New}          % Create new stream tail
      thread Old=Message|New end    % Add message to stream
   end
end
```

[11]This definition maintains order between causally-related sends. The definition can be extended to maintain order only within threads by using queues indexed by thread name.

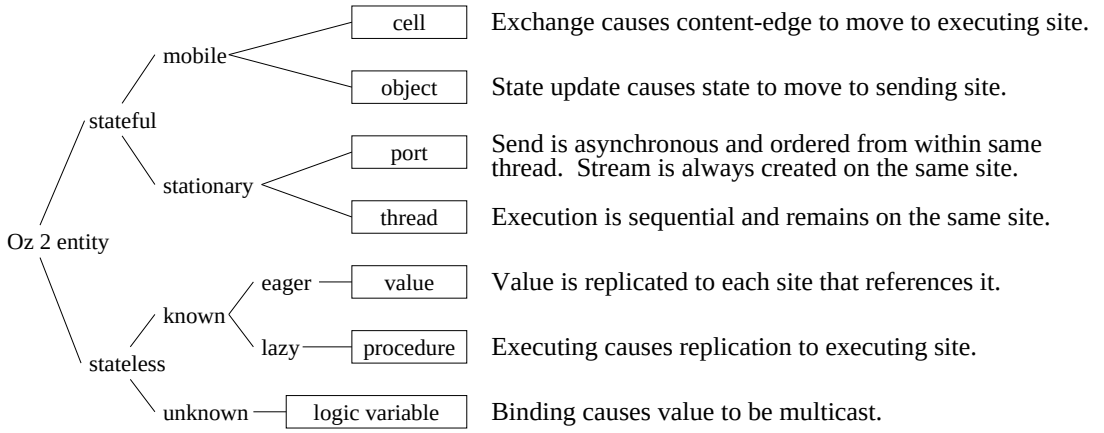


Figure 7.3: Oz 2 entities and their distributed semantics.

Calling `{NewPort Str Snd}` creates the procedure `Snd` and ties it to the stream `Str`. Calling `{Snd X}` appends `X` to the stream and creates a new unbound end of the stream. That is, the tail `S` of the stream is bound (`S=X|S2`), and `S2` becomes the new tail. One way to build a *server* is to create a thread that waits until new information appears on the stream and then takes action, depending on this information. Section 7.6.4 shows how to use a port to code a server.

7.6 Distribution Model

Distributed Oz defines a distributed semantics for all basic entities and compound entities. By this we mean that each operation of each entity is given a well-defined network behavior. This section defines and motivates this behavior. The distributed semantics for all seven entities is classified and summarized in Figure 7.3. We discuss separately replicable entities (values and procedures), logic variables, and stateful entities (cells, objects, ports, and threads). The semantics of the basic stateful entity, the cell, are defined in Section 7.7. Appendix 7.11.1 proves correctness of the cell's mobile state protocol. Definitions and correctness proofs for objects and ports should be easy to devise after understanding the cell.

7.6.1 Replication

All stateless entities, namely values and procedures, are replicated. That is, copies are made when necessary to improve network behavior. Since the entities never change, using copies does not affect the language semantics. An important design decision is how much of a stateless data structure to replicate eagerly. Too much may cause resource limits to be exceeded on the receiving site. Too little may cause a great increase in latency. The current design incorporates both eagerness and laziness so that the programmer can program the degree of laziness desired.

We summarize the *lazy replication* protocol, i.e., the distributed algorithm that manages replication. Records, numbers, and literals are replicated eagerly, i.e., there is no such thing as a remote reference to a record.[12] Procedures are replicated lazily, i.e., one may have remote references to a procedure. The procedure is replicated when it is applied remotely. Both the compiled code and the closure are given global addresses. Therefore a site has at most one copy of each code block and closure. All network messages, except for responses to explicit replication requests, do not contain any procedure code nor closure. A replication request is sent only if the code or closure is not available locally.

The two extremes (eager and lazy) are thus provided. This allows the programmer to design the degree of eagerness or laziness desired in his or her data structures. This is not the only possible design; good arguments can also be given for eager procedure replication and an independent mechanism to introduce laziness.

The following example shows how a large tree can be dynamically partitioned into eager and lazy subtrees by introducing procedures:

```
proc {MakeLazyTree E L R ?X}
I1 I2 in
   X=bigtree(leftlazy:I1 rightlazy:I2 eagerbranch:E)
   proc {I1 T1} T1=L end
   proc {I2 T2} T2=R end
end
```

Executing `{MakeLazyTree E L R X}` with three record arguments `E`, `L`, and `R` returns a record `X` with one eager field corresponding to `E` and two lazy fields corresponding to `L` and `R`. When a reference to `X` is communicated to a site, the subtree at `eagerbranch` is transferred immediately while the subtrees at `leftlazy` and `rightlazy` are not transferred. Executing `{I1 Y}` transfers the left subtree and binds it to `Y`.

7.6.2 Logic Variables

A logic variable is a reference to a value that is not yet known. There are two basic operations on a logic variable: binding it and waiting for it to have a value. Binding a logic variable eagerly replaces the logic variable on all sites that reference it. Since a value is stateless, any number of readers can wait concurrently for the value to become known. This binding protocol, also called *variable elimination* protocol, is the only distributed algorithm needed to implement distributed unification.[13]

In addition to their role in improving latency tolerance, logic variables provide the programmer with an efficient and expressive way to dynamically manage multicast groups. A *multicast* sends data to a predefined subset of all network addresses, called the multicast group. Recent protocol designs support multicast as a way to increase

[12] Therefore, it is possible to have multiple copies of the same record on a site, since the same record may be transferred many times.

[13] The distributed unification algorithm will be the subject of another article.

network efficiency [32]. Binding a logic variable can be implemented using a multicast group. Binding to a record multicasts the record. Binding to a procedure multicasts only the name of the procedure (not the closure). Binding to another logic variable merges the two multicast groups.

If a logic variable is bound to a list whose tail is a logic variable, then a new multicast group can be immediately created for the tail. Implementations may be able to optimize the binding to reuse the multicast group of the original logic variable for the tail. In this way, efficient multicasting of information streams can be expressed transparently.

7.6.3 Mobility Control

Any stateful entity, i.e., cell, object, port, or thread, is characterized by its home site. Mobility control defines what happens to the home site for each operation on the stateful entity. For example, invoking an exchange operation on a cell is defined to change the home site to be the invoking site. A stateful entity is referred to as mobile or stationary, depending on whether the entities' basic state-updating operation is mobile or stationary.

To allow programming of arbitrary communication patterns, the language must provide at least one mobile and one stationary entity. To satisfy this condition, we define cells and objects to be mobile and ports and threads to be stationary.

— **Cells.** A cell is mobile. A cell may be accessible from many sites, each of which knows the cell name. Only the home site contains the cell's content-edge, which pairs the name and the content. Invoking an exchange from any site causes a synchronous move of the content-edge to that site. This is done using a mobile state protocol to implement the interleaving semantics of the exchanges (see Section 7.7). Invoking a cell-access operation does not move the content-edge. Only the cell's content is transferred to the invoking site.

— **Objects.** An object is mobile, and its distribution semantics obeys its OPM definition. When a method is called remotely, the procedures corresponding to the object and the method are replicated to the calling site. The method is then executed at the calling site. When the object's state is updated, the content-edge of the cell holding the state will migrate to the site. Subsequent method calls will be completed locally without network operations. If the object possesses a lock, then operations on the object's state will not be disturbed by remote requests for the lock until it is released. This is implemented by using a cell-access operation to check the current thread (see reentrant locks).

— **Ports.** A port is stationary. Invoking a send from any site causes a new entry to appear eventually in the port's stream at its home site. The send operation is defined to be *asynchronous* (nonblocking) and *ordered* (FIFO). This behavior cannot be defined in terms of OPM. It is proper to the distributed semantics.

```
proc {MakeStat Obj ?StatObj}
   Str
   proc {Serve S}
      if M Ss in S=M|Ss then
         {Obj M}
         {Serve Ss}            % Loop on received
messages
      else skip end
   end
in
   {NewPort Str StatObj}       % Port is stationary
   thread {Serve Str} end      % The server loop
end
```

Figure 7.4: Making an object stationary (first attempt).

Send operations complete immediately (independently of any network operations), and messages sent from a given thread appear on the stream in the same order that they were sent.

We provide ports as an entity for two reasons. First, a stationary port is a natural way to build a server. Second, the asynchronous ordering supports common programming techniques and can exploit popular network protocols. Ports have a second operation, localize, which causes a synchronous move of the home site to the invoking site. Without the ability to localize ports, mostly stationary objects cannot be implemented transparently (see Section 7.6.4).

— **Threads.** A thread is stationary. The reduction of a thread's statement is done at its home site, which is the thread's creation site. A thread cannot change sites. First-class references to threads may be passed to other sites and used to control the thread. For example, an exception may be raised in a thread from a remote site. Commands to a thread from a remote site are sent synchronously and in order, as an RPC. This is modeled as if the target thread access were packaged in a port, and the calling thread suspends until the operation is performed.

The protocols used to implement mobility control for objects and ports are both based on the mobile state protocol given in this article. They are extended to provide for the operations of the particular entity. For example, the port protocol manages the mobility of the home site as well as the FIFO connections from other sites to the home site. The port protocol is defined in Section 7.7.4.

7.6.4 Programming with Mobility Control

We show how to concisely program arbitrary migratory behavior using the entities of Distributed Oz. Expressing other distributed-programming idioms (e.g., RPC and

client-server architectures) is left as an exercise for the reader. We assume the existence of primitives to initiate a computation on a new site.

In a user program, the mobility of an object must be well-defined and serve the purpose of the program. Some objects need to stay put (e.g., servers), and others need to move (e.g., mobile agents or caches). The most general scenario is that of the caller and the object negotiating whether the object will move. The basic implementation technique is to define procedures to transparently limit the mobility of an object, which is freely mobile by default. We show in three steps how this is achieved:

- **Freely mobile objects.** This is the default behavior for objects. Any object defined as in Section 7.5.2 will move to each site that sends a message that updates the object's state. The object is guaranteed to stay at the site until the lock is released within the invoked method. While the lock is held, the object will not move from the site.

- **Stationary objects.** A stationary object executes all its methods on the same site. Any object can be made stationary using the technique given in Section 7.6.4.

- **Mostly stationary objects.** A mostly stationary object remains stationary unless explicitly moved. Any freely mobile object can be made mostly stationary using the technique given in Section 7.6.4.

Each of the latter two cases defines a procedure that can control the mobility of *any* mobile object. This is an important modularity property: it means that one can change the network behavior of a program's objects without rewriting the objects in any way. The objects and their mobility properties are defined independently. This property is obtained independently of the object system's metaobject protocol.

Stationary Objects

An object can be made stationary be wrapping it in a port. Figure 7.4 shows a simple way to do this by defining the procedure `MakeStat` that takes any object `Obj` and returns the procedure `StatObj`. The result of calling `StatObj` is to send messages to a port. The thread in the construction **`thread`** `{Serve Str}` **`end`** is responsible for the actual object invocation. The thread takes messages from the port's stream and sends them to the object. The thread does not move. Therefore, `StatObj` behaves like `Obj` except that `Obj` does not move. After the first object invocation, the object is at the site of the server loop and will not move (unless, of course, some threads are given direct references to `Obj`). For example, if upon its creation `Obj` is passed directly to `MakeStat`, and only references to `StatObj` are passed to others, then `Obj` will forever remain on its creation site.

However, the solution in Figure 7.4 is too simple. `StatObj` deviates from providing *exactly* the same behavior as `Obj` in three ways. First, sending messages to `StatObj` is asynchronous, whereas sending messages to `Obj` is synchronous. Second, only one method of `Obj` can be executing at a time, since `Obj` is inside its own thread. Third, exceptions in `Obj` are not seen by `StatObj`.

```
      proc {MakeStat Obj ?StatObj}
         Str Send
         proc {Serve S}
            if M Sync Ss in S=msg(M Sync)|Ss then
               thread
                  try {Obj M} Sync=unit
                  catch E then Sync=exception(E) end
               end
               {Serve Ss}            % Loop on received
messages
            else skip end
         end
      in
         {NewPort Str Send}          % Port is stationary
         proc {StatObj M}
         Sync E in
            {Send msg(M Sync)}       % Sync variable ensures
order
            if Sync = exception(E) then
              raise E end
            else skip end
         end
         thread {Serve Str} end      % The server loop
      end
```

Figure 7.5: Making an object stationary (correct solution).

Figure 7.5 gives a correct definition of the procedure `MakeStat`. The logic variable `Sync` is used to synchronize `{StatObj M}` with `{Obj M}`. The notation `msg(M Sync)` pairs `M` and `Sync` in a record. Waiting until `Sync` is bound to a value guarantees that the thread executing `{StatObj M}` continues execution only after `{Obj M}` is finished. This means that messages sent from within one thread are received in the order sent. The example also models exceptions correctly by transferring the exceptions back to the caller.[14]

There are many useful variations of this solution:

- Leaving out the synchronization variable `Sync` makes `StatObj` behave asynchronously. Then message sending is a local operation that takes constant time. Messages are received in any order.

- Leaving out the **`thread ... end`** inside `Serve` ensures that `Obj` executes only a single message at a time. Together with the synchronization variable, this results in an end-to-end flow control between the sender and receiver. All messages sent are serialized at `Obj` and only a single message is handled at a time.

This solution makes it clear that a stationary object is not a simple concept. It requires synchronization between sites, passing of exceptions between sites, and thread creation. Freely mobile objects are simpler, since their execution is always local. The mechanics of making an object stationary can be encapsulated, as is done here, to hide its complexity from the user. In general, most of the complexity of concurrent programming can be encapsulated. The practicality of this approach is demonstrated on an industrial scale by the Ericsson Open Telecom Platform [11, 36].

Mostly Stationary Objects

It may be desirable in special cases to move an object that has been made stationary, e.g., for a server to move closer to its clients or to leave a machine that will be shut down. We require a solution in which it is eventually true that sending a message will need only a single network hop, no matter how many times the object moves. Figure 7.6 gives a solution that uses the localize operation for ports. Port mobility is derived from cell mobility; see Sections 7.6.3 and 7.7.4. The figure defines the procedure `MakeFirm` that takes any freely mobile object `Obj` and returns two procedures, `FirmObj` and `Move`. The procedure `FirmObj` has identical language semantics to `Obj` but is stationary.[15] `Move` is a zero-argument procedure that when applied, atomically moves the object to the invoking site. That is, in each thread's stream of messages to `FirmObj`, there is a point such that all earlier messages are received on the original site, and all later messages are received on the new site.

[14] One subtle point remains that requires attention. The encapsulated object can escape by a method returning **`self`**. The problem can be solved by using inheritance to make the object stationary or by using the Oz 2 metaobject protocol [60]. We do not show the details, since this would take us too deep into the object system.

[15] Adding the corrections of Figure 7.5 is left as an exercise for the reader.

```
      proc {MakeFirm Obj ?FirmObj ?Move}
         Str Prt Key={NewName $}
         proc {Serve S}                    % Stoppable server
proc.
            if Stopped Rest Ss in
               S=Key(Stopped Rest)|Ss then
               Rest=Ss
               Stopped=unit
            elseif M Ss in S=M|Ss then
               {Obj M}
               {Serve Ss}
            else skip end
         end
      in
         proc {Move}
         Stopped Rest in
            {Prt Key(Stopped Rest)}        % Stop old server
loop
            {Wait Stopped}
            {Localize Prt}                 % Transfer to new
site
            thread {Serve Rest} end        % Start new server
loop
         end
         {NewPort Str Prt}
         proc {FirmObj M} {Prt M} end
         thread {Serve Str} end            % Initial server
loop
      end
```

Figure 7.6: Making an object mostly stationary.

Like a stationary object, a mostly stationary object consists of an object wrapped in a port. To handle incoming messages, a server loop is installed on the port's home site. To move the object, this loop is stopped; the port's home site is moved using the localize operation; and a new server loop is installed on the new site. The server loop in Figure 7.6 is stopped by sending the message `Key(Stopped Rest)`, where `Key` is an Oz 2 name used to identify the message and where `Stopped` and `Rest` are outputs. Since `Key` is unforgeable and known only inside `MakeFirm`, the server loop can be stopped only by `Move`. The port `Prt` must be hidden inside a procedure; otherwise it can be localized by any client. When the loop is stopped, `Rest` is bound to the unprocessed remainder of its message stream. The new server loop uses `Rest` as its input.

From an algorithmic viewpoint, Figure 7.6 defines the distributed algorithm for mostly stationary objects. This algorithm is a composition of three simpler algorithms: (1) a mobile state protocol (for cells and extended for ports), which is the scope of this article, (2) a variable elimination protocol (see Section 7.6.2), and (3) a lazy replication protocol (see Section 7.6.1). The three algorithms are composed by means of a notation which is exactly the OPM language. The technique of factoring complex algorithms into simpler components has many advantages. For example, Figure 7.6 can be extended in a straightforward way with a failure model, using the exceptions of OPM.

7.7 Cells: Semantics and Mobile State Protocol

In this section we specify the language semantics and distributed semantics of cells. We first define both semantics in a high-level manner. In general, the language semantics is defined as a transition relation between configurations. A configuration is a pair consisting of a statement and a store. The distributed semantics is an orthogonal refinement of the language semantics where the notion of site is made explicit. It is carefully designed to give a simple programmer model of network awareness. In this article, however, we confine ourselves to the model of cell mobility.

It will be shown in the case of the cell exchange operation that the distributed semantics correctly implements the language semantics, hence achieving network transparency. The mobile state protocol is part of a graph model of the execution of OPM. We give the graph model in just enough detail to set the context of the protocol. We then give an informal description and a formal specification of the protocol. Appendix 7.11.1 proves that the formal specification correctly implements the distributed semantics and consequently the language semantics.

7.7.1 Cell Semantics

Among the basic operations on cells there are creation, exchange, and access. We give the language semantics of these operations and the distributed semantics of exchange. The distributed semantics of the other operations should be relatively easy to devise

after understanding the exchange.

Basic Notation

All execution is described by the reduction of transition rules. The reduction is an atomic operation that is described by a rule written according to the following diagram:

$$\frac{(\textit{statement}) \;\|\; (\textit{new statement})}{(\textit{store}) \;\|\; (\textit{new store})}$$

This rule becomes applicable for a given statement when the actual store matches the store given in the rule. Because the language is concurrent, reduction is in general nondeterministic. The effect of a reduction is to replace the current configuration by a *set* of result configurations. In the case of cell operations, this set is always a singleton. Fairness between threads implies that a rule is guaranteed to reduce eventually when it is applicable and when it refers to the first statement of a thread.

Language Semantics

We give the transition rules for cell creation, access, and exchange. For all rules, the part of the store that is not relevant to the rule is denoted by σ.

Cell creation $\dfrac{\texttt{\{NewCell X C\}} \;\|\; \texttt{C=n}}{\sigma \;\|\; \sigma \wedge \texttt{n:X}}$ newName(n)

Cell creation is provided by the operation `{NewCell X C}`. The statement reduces to the new statement `C=n`, where `n` is a new name taken from an infinite set of fresh names. The new store contains the content-edge `n:X`, which pairs `n` with the initial content `X`.

Cell access $\dfrac{\texttt{\{Access C X\}} \;\|\; \texttt{X=Z}}{\sigma \wedge \texttt{C=n} \wedge \texttt{n:Z} \;\|\; \sigma \wedge \texttt{C=n} \wedge \texttt{n:Z}}$

Cell access is provided by the operation `{Access C X}`. The rule is reducible when its first argument refers to a cell name. It reduces to the binding `X=Z`. The store is unchanged. The binding is defined through other reduction rules. Reducing the binding gives access to the content `Z` through `X`. If `X` and `Z` are incompatible, then an exception is raised.

Cell exchange $\dfrac{\texttt{\{Exchange C X Y\}} \;\|\; \texttt{X=Z}}{\sigma \wedge \texttt{C=n} \wedge \texttt{n:Z} \;\|\; \sigma \wedge \texttt{C=n} \wedge \texttt{n:Y}}$

Cell exchange is provided by the operation `{Exchange C X Y}`. The rule is reducible when its first argument refers to a cell name. It reduces to the new statement `X=Z`, which gives access to the old content `Z` through `X`. The content-edge is updated to refer to the new content `Y`.

Distributed Semantics of Exchange

The distributed semantics is an extension of the language semantics that specifies how the reduction is partitioned among multiple sites. We introduce the notion of a *representative* on a site. This notion is used to place statements and the store contents on one or more sites. By store contents we mean content-edges and references to values or logic variables. The representative of X on site i is denoted by X_i. The subscript denotes the site. The exact notion of representative depends on what Oz 2 entity is considered (see Figure 7.3). In the case of cells it is defined here. For other entities it is straightforward to devise after understanding the language graph and distribution graph defined in Section 7.7.2. We define the distributed semantics in three steps:

- Define the representation of an entity as a formula written in terms of representatives.
- Define a mapping M from the representation of an entity to the entity it represents.
- Formulate the reduction rules in terms of representations.

To show that the distributed semantics implements the language semantics, we use a technique variously known as *abstraction* [77] or *simulation* [85]. Consider the configuration (S, σ). Its reduction gives a set of new configurations (S', σ'). Consider the corresponding configuration (S_r, σ_r), written in terms of representatives. Its (distributed) reduction gives a set of (S'_r, σ'_r). This is summarized in the following diagram:

$$\begin{array}{ccc} (S,\sigma) & \xrightarrow{ls} & (S',\sigma') \\ M \uparrow & & \uparrow M \\ (S_r,\sigma_r) & \xrightarrow{ds} & (S'_r,\sigma'_r) \end{array} \tag{7.1}$$

To show that the distributed semantics *ds* implements the language semantics *ls*, we must show that M applied to the set of possible configurations (S'_r, σ'_r) gives the same result as the set of possible configurations (S', σ').

Assume that a cell with name *n* exists on sites 1, ... *k* and that the content-edge is on site *p* with content Z. We define the mapping M as follows:

Distributed semantics	Language semantics
$\mathtt{C}_1\mathtt{=n} \wedge ... \wedge \mathtt{C}_k\mathtt{=n}$	C=n
$(\mathtt{n:Z})_p \wedge 1 \leq \forall i \leq k, i \neq p : (\mathtt{n}:\bot)_i$	n:Z

The cell is accessible from sites 1, ..., *k*, so that $\mathtt{C}_1$=n, $\mathtt{C}_2$=n, ... $\mathtt{C}_k$=n together imply C=n. The content-edge on site *p* is denoted by $(\mathtt{n:Z})_p$. The other sites know the cell but do not have the content-edge. This is denoted by $(\mathtt{n}:\bot)_i$ for $i \neq p$. If we assume the exchange is invoked on site *q* and that the content-edge is on site *p*, where $1 \leq p, q \leq k$, then the distributed reduction rule is

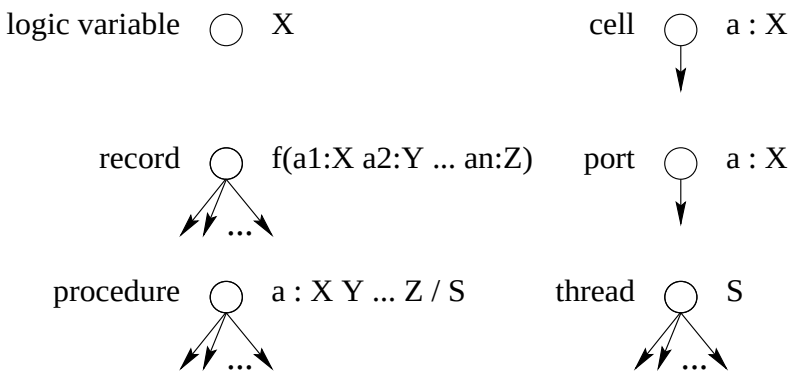


Figure 7.7: Nodes in the language graph.

$\{\texttt{Exchange C X Y}\}_q$	$(\texttt{X=Z})_q$
$\sigma_r \wedge \texttt{C}_1\texttt{=n} \wedge \ldots \wedge \texttt{C}_k\texttt{=n} \wedge$ $(\texttt{n:Z})_p \wedge 1 \leq \forall i \leq k, i \neq p : (\texttt{n:}\bot)_i$	$\sigma_r \wedge \texttt{C}_1\texttt{=n} \wedge \ldots \wedge \texttt{C}_k\texttt{=n} \wedge$ $(\texttt{n:Y})_q \wedge 1 \leq \forall i \leq k, i \neq q : (\texttt{n:}\bot)_i$

We assume that the representatives $\texttt{X}_q$, $\texttt{Y}_q$, and $\texttt{Z}_q$ are created if needed to complete the reduction. From this rule it follows that the cell is accessible from multiple sites, that the content-edge exists on exactly one of these sites, that the exchange is performed on exactly one of these sites, and that after the exchange the content-edge is on the same site as the exchange.

Theorem 1 *(T1).* The distributed semantics of exchange implements the language semantics of exchange.

Proof. Consider the reduction performed by the rule for distributed exchange. It is clear from inspection that M continues to hold after the reduction. □

If multiple exchanges are invoked on site q, it is easy to see that if $p \neq q$ then the first exchange requires nonlocal operations. One can deduce also that subsequent exchanges are purely local. If exchanges are invoked from many sites, then they will be executed in some order. If the content-edge refers to an object state, then the object, while mobile, will be correctly updated as it moves from site to site. The system uses a *mobile state protocol* to implement this rule.

7.7.2 The Graph Model

We present a graph model of the distributed execution of OPM. The graph model plays a key role in bridging the distributed semantics with the mobile state protocol and the implementation architecture. An OPM computation space can be represented in terms of two graphs: a *language graph*, in which there is no notion of site, and a *distribution graph*, which makes explicit the notion of site. We explain what mobility means in terms of the distribution graph. Finally, we summarize the failure model in terms of the distribution graph.

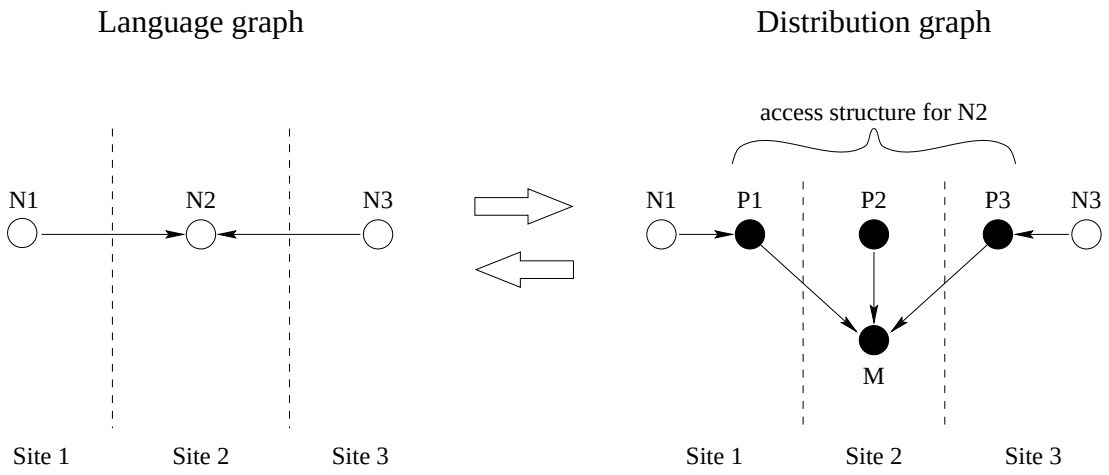


Figure 7.8: From language graph to distribution graph.

The Distribution Graph

The distributed execution of OPM is introduced in two steps. In the first step, we model an OPM computation space as a graph, called *language graph*. Each Oz 2 entity except for an object corresponds to one node in the language graph (see Figure 7.7). An object is a derived concept that is modeled as a subgraph, namely a procedure with references to the object's state, lock, and methods. OPM execution is modeled as a sequence of graph transformations.

In the second step, we extend the language graph with the notion of *site*. First introduce a finite set of sites, and then annotate each node of the language graph with a site. If a node is referenced by a node on another site, then map it to a *set* of nodes. This set is called the *access structure* of the original node (see Figure 7.8). An access structure consists of a set of *global nodes*, namely one *proxy node* per site and one *manager node* for the whole structure. In a cell access structure, the content-edge is an edge from exactly one proxy node to the content. Nodes that are only accessed locally (*local nodes*) do not have an access structure. In this case, the content-edge is an edge from the local node to the content.

The graph resulting after all nodes have been transformed is called the *distribution graph*. OPM execution is again modeled as a sequence of graph transformations. These transformations respect language semantics while defining the distributed semantics. For a cell access structure, a proxy node P_i on site i corresponds to the representatives $\mathtt{c}_i\mathtt{=n} \wedge (\mathtt{n:x})_i$ if the content-edge is on site i, and otherwise to $\mathtt{c}_i\mathtt{=n} \wedge (\mathtt{n:}\bot)_i$. The content-edge itself corresponds to the representative $(\mathtt{n:x})_i$.

Mobility in the Distribution Graph

At this point, it is useful to clarify how cell mobility fits into the distribution graph model. First, the nodes of the distribution graph never change sites. A manager node has a global address that is unique across the network and never changes. This makes

memory management very simple, as explained in Section 7.8.2. Second, access structures can move across the network (albeit slowly) by creating proxies on fresh sites and by losing local references to existing proxies. Third, a content-edge can change sites (quickly) if requested to do so by a remote exchange. This is implemented by a change of state in the cell proxies that is coordinated by the mobile state protocol.

The mobile state protocol is designed to provide efficient and predictable network behavior for the common case of no failure. It would be extremely inefficient to inform all proxies each time the content-edge changes site. Therefore, we assume that proxies do not in general know where the content-edge is located. A proxy knows only the location of its manager node. If a proxy wants to do an exchange operation, and it does not have the content-edge, then it must ask its manager node. The latency of object mobility is therefore at most three network hops (less if the manager node is at the source or destination).

Having a fixed manager node greatly simplifies the implementation. However, it reduces locality and introduces an unwanted dependency on a third party (i.e., the manager site). For example, object movement within Belgium is expensive if the manager is in Sweden, and it becomes impossible if the network connection to Sweden breaks down. We present two possible extensions to the mobile state protocol, each of which solves these problems and is compatible with the current system architecture. The final solution is being designed in tandem with the failure model (see below). The first solution is to dynamically construct a tree of managers, such that each proxy potentially has a manager on its own site. The second solution is for the proxies to change managers. For example, assume the old manager knows all its proxies. To change managers, it creates a new manager and then it sends a message to each proxy informing it of the new manager.

The Failure Model

The failure model must reliably inform the programmer if and when a failure occurs and allow him or her to take the necessary actions. The failure model is still under discussion, so the final design will likely differ from the one presented here. This section summarizes an extension to the mobile state protocol that provides precise failure detection. The programmer can enable a failure to appear in the language as an exception. Objects based on the extended protocol can be used as building blocks to program reliable objects. For example, one can program a reliable cell that has a primary-slave architecture [29].

We distinguish between network failure and site failure. All failures become visible lazily in a proxy. For sites we assume a stopping failure (crash) model: a failed site executes correctly up to the moment of the failure, after which it does nothing. A thread attempts to access a proxy to do a cell operation. If the access structure cannot continue to function normally, then the proxy becomes a failure node, and attempts to invoke an operation can cause an exception to be raised in the thread.

In the case of site failure, a cell access structure has two failure modes:

- **Cell failure**. The complete access structure fails if either the manager node

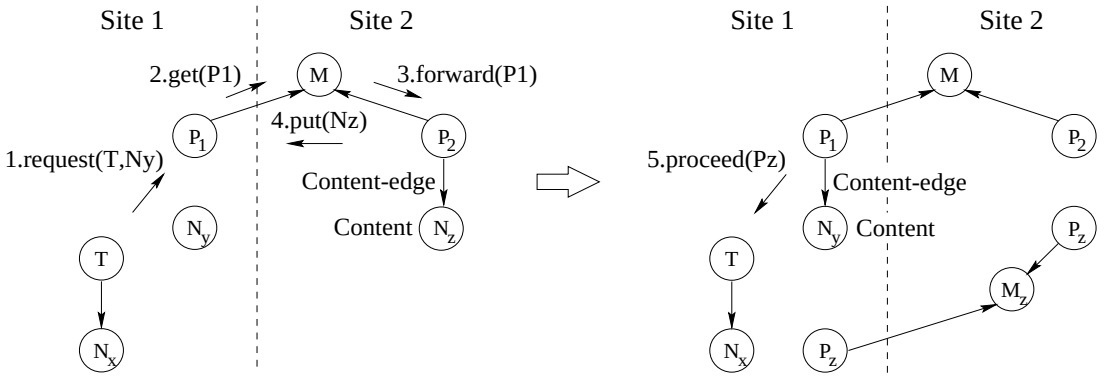


Figure 7.9: Exchange initiates migration of content-edge.

fails or if a proxy that contains the content-edge fails. The manager does not know at all times precisely where the content-edge is. The manager bounds the set of proxies that may contain the content-edge by maintaining a conservative approximation to the *chain* structure (see Appendix 7.11.1). The content-edge is guaranteed to be in the chain. If one proxy in the chain fails, then the manager interrogates the proxies in the chain to distinguish between cell failure and proxy failure.

- **Proxy failure**. This happens if a proxy fails that does not contain the content-edge. This does not affect the computation and may be safely ignored.

It is impossible in general to distinguish between a failed site and a very slow network. A cell may therefore fail even if no site has failed. This will normally be a rare event.

7.7.3 Informal Description

We first give an informal description of the mobile state protocol. The protocol is defined with respect to a single cell. Assume that the cell is accessible from a set of sites. Each of these sites has a proxy node responsible for the part of the protocol on that site. The proxy node is responsible for all cell behavior visible from its site. In addition, there is a single manager node that is responsible for coordinating the proxy nodes. These nodes together implement the distributed semantics of one cell.

The content-edge is stored at one of the cell proxies. Cell proxies exchange messages with threads in the engine. To ask for the cell content, a thread sends a message to a proxy. The thread then blocks waiting for a reply. After executing its protocol, the proxy sends a reply giving the content. This lets the thread do the binding. Figure 7.9 shows how this works. We assume that the content-edge is not at the current proxy. A proxy requests the content-edge by sending a message to the manager. The manager serializes possible multiple requests and sends forwarding commands to the proxies. The current location of the content-edge may lag behind the manager's knowledge of who is the eventual owner. This is all right: the content-edge will eventually be forwarded to every requesting site.

Many requests may be invoked concurrently to the same and different proxies, and the protocol takes this into account. A request message from a thread that issued `{Exchange C X Y}` will atomically achieve the following results: the content `Z` is transferred to the requesting site; the old content-edge is invalidated; a new content-edge is created bound to `Y`; and the bind operation `X=Z` becomes applicable in the requesting thread.

Messages. The protocol uses the following nodes and messages. P_i denotes the addresses of proxies in the distribution graph corresponding to cell `C`. N_X, N_Y, N_Z denote the addresses of nodes corresponding to variables `X`, `Y`, and `Z`. A manager understands **get(P)**. A proxy understands **put(N)**, **forward(P)**, and **request(T,N)**, where T is the requesting thread. A thread understands **proceed(N)**.

Outline of protocol (Figure 7.9).

1. Proxy P_1 receives a **request(T,N_Y)** from the engine. This message is sent by thread T as part of executing `{Exchange C X Y}`. Thread T blocks until the proxy replies. N_Y is stored in P_1 (but does not yet become the content-edge). If the content-edge is at P_1 and points to some node N_a, then P_1 immediately sends **proceed(N_a)** to T. Otherwise, **get(P_1)** is sent to the manager.

2. Manager M receives **get(P_1)**. Manager sends **forward(P_1)** to the current owner P_2 of the content-edge and updates the current owner to be P_1.

3. Proxy P_2 receives **forward(P_1)**. If P_2 has the content-edge, which points to N_Z, then it sends **put(N_Z)** to P_1 and invalidates its content-edge. Otherwise, wait until the content-edge arrives at P_2. Sending the message **put(N_Z)** causes the creation of a new access structure for N_Z.[16] From this point onward, all references to N_Z are converted to P_Z.

4. Proxy P_1 receives **put(P_Z)**. At this point, the content-edge of P_1 points to N_Y. P_1 then sends **proceed(P_Z)** to thread T.

5. Thread T receives **proceed(P_Z)**. The thread then invokes the binding of N_X and P_Z.

7.7.4 Formal Specification

We formally define the mobile state protocol as a set of nondeterministic reduction rules that determine the behavior of a subset of the distribution graph. We assume nodes of three types: proxy, manager, and thread nodes. The proxy and manager nodes form an access structure for a cell. We first define the notation for the rules and the nodes' internal state. Then we give the rule definitions. Finally, we describe how to extend the protocol for ports. Appendix 7.11.1 gives a formal proof that the protocol correctly implements the language semantics of cells.

[16] For all types of entities N_Z except records, which are replicated eagerly.

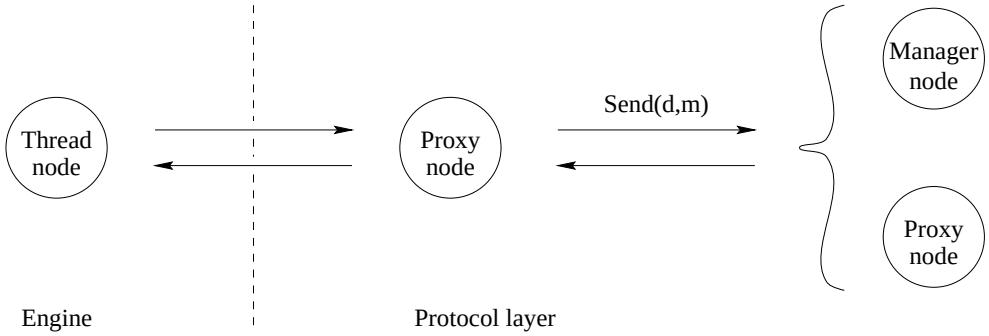


Figure 7.10: Interface between engine and protocol.

Preliminaries

Consider a single manager node M, a set of k proxy nodes P_i with $1 \leq i \leq k$, and a set of m thread nodes T_i with $1 \leq i \leq m$. All nodes have state, can send messages to each other according to Figure 7.10, and can perform internal operations. Let these nodes be linked together by a network N that is a multiset containing messages of the form $d : m$ where d identifies a destination (proxy, manager, or thread node) and where m is a message.

The protocol is defined using reduction rules of the form

$$\frac{\text{Condition}}{\text{Action}} \quad .$$

Each rule is defined in the context of a single node. Execution follows an interleaving model. At each reduction step, a rule with valid condition is selected. Its associated actions are reduced atomically. A rule condition consists of boolean conditions on the node state and one optional receive condition **Receive**(d,m). The condition **Receive**(d,m) means that $d : m$ has arrived at d. Executing a rule with a receive condition removes $d : m$ from the network and performs the action part of the rule. A rule action consists of a sequence of operations on the node state with optional sends. The action **Send**(d,m) asynchronously sends message m to node d, i.e., it adds the message $d : m$ to the network. The action **Receive**(d,m) blocks until message m arrives at d, at which point it removes $d : m$ from the network and continues execution.

We assume that the network and the nodes are *fair* in the following sense. The network is asynchronous, and messages to a given node take arbitrary finite time and may arrive in arbitrary order. All rules that are applicable infinitely often will eventually reduce.

Node State

The node state is represented as a set of attributes of each node. Table 7.3 lists the attributes for proxy, manager, and thread nodes, along with their initial values. This table assumes without loss of generality that the content-edge is initially at proxy P_1. A *NodeRef* is a reference to any node. GetManagerRef() returns a reference to the

Thread T_i		
Attribute	Type	Initial value
id	*NodeRef*	GetThreadRef(*i*)
Manager M		
Attribute	Type	Initial value
tail	*NodeRef*	GetProxyRef(1)
Proxy P_i		
Attribute	Type	Initial value
state	{FREE,CHAIN}	CHAIN ($i = 1$), FREE ($i \neq 1$)
content	NULL | *NodeRef*	N ($i = 1$), NULL ($i \neq 1$)
forward	NULL | *NodeRef*	NULL
thread	NULL | *NodeRef*	NULL
newcontent	NULL | *NodeRef*	NULL
manager	*NodeRef*	GetManagerRef()
id	*NodeRef*	GetProxyRef(*i*)

Table 7.3: Node State

manager M. GetProxyRef(*i*) returns a reference to proxy P_i. GetThreadRef(*i*) returns a reference to thread T_i. N is a *NodeRef* giving the initial content of the cell. NULL is a special value that marks an attribute as not valid. P.manager, P.id, and T.id are constants.

Rule Definitions

The protocol is defined in two parts. Figure 7.11 defines the procedure exchange_and_bind, which is part of the thread. Figure 7.12 defines the protocol rules. Rules 1–5 are defined for proxy nodes. Rule 6 is defined for the manager node.

The reduction of both the statement `{Exchange C X Y}` and its resulting binding `X=Z` is implemented in thread T by calling exchange_and_bind(P,N_X,N_Y). The nodes P, N_X, and N_Y correspond to the variables `C`, `X`, and `Y`. T sends a message containing N_Y to the proxy P on its site. T blocks until the content-edge and the content have moved to its site. Then the proxy sends the old content N_Z to T. The exchange is completed, strictly speaking, when N_Z is received in T, because that is when the bind operation becomes applicable. The thread then invokes bind(N_X,N_Z).

Appendix 7.11.1 gives a proof that this specification implements the distributed semantics of exchange. Every exchange eventually results in a bind operation with correct arguments, and the content is updated correctly in the access structure. Exchange requests on a site without the content-edge will invoke the bind operation when the content arrives.

P.state=FREE if P has not requested the content-edge. P.state=CHAIN if P has requested the content-edge, which may not have arrived yet. P.content≠NULL if and only if P has the content-edge. P.forward≠NULL if and only if P should forward the content when it is present. P.thread and P.newcontent are used when the content-edge is

```
procedure exchange_and_bind(P,N_X,N_Y)
   Send(P,request(T.id,N_Y))
   Receive(T,proceed(N_Z))
   bind(N_X,N_Z)
end
```

Figure 7.11: Mobile state protocol: Thread interface.

remote. They store the information necessary for a correct reply when the content-edge arrives locally.

Extension for Port Mobility

The port protocol is an extension of the cell protocol defined in the previous section. As explained in Section 7.6.3, a port has two operations, send and localize, which are initiated by a thread referencing the port. The localize operation uses the same protocol as the cell exchange operation. For a correct implementation of the send operation, the port protocol must maintain the FIFO order of messages even during port movement. Furthermore, the protocol is defined so that there are no dependencies between proxies when moving a port. This means that a single very slow proxy cannot slow down a localize operation.

Each port home is given a *generation identifier*. When the port home changes sites, then the new port home gets a new generation identifier. Each port proxy knows a generation which it believes to be the generation of the current port home. No order relation is needed on generations. It suffices for all generations of a given port to be pairwise distinct. For simplicity they can be implemented by integers.

The send operation is asynchronous. A send operation causes the port proxy to send a message to the port home on a FIFO channel. The message is sent together with the proxies' generation. If a message arrives at a node that is not the home or has the wrong generation, then the message is bounced back to the sending proxy on a FIFO channel. If a proxy gets a bounced message then it does four things. It no longer accepts send operations. It then asks the manager where the current home is. When it knows this, it recovers all the bounced messages in order and forwards them to the new home. Finally, when it has forwarded all the bounced messages, it again accepts send operations from threads on its site.

7.8 System Architecture

The mobile state protocol defines the distributed semantics of cells. This protocol is only one aspect of a Distributed Oz implementation. Other important aspects include the interface between the protocol and the centralized execution, how the protocol is built on top of a shared computation space, and how the shared computation space

1. Request content (content not present).

Receive(P,request(T,N_y)) $\wedge$ P.state=FREE

Send(P.manager,get(P.id))
P.state $\leftarrow$ CHAIN
P.thread $\leftarrow$ T
P.newcontent $\leftarrow$ N_y

2. Request content and reply to thread (content present).

Receive(P,request(T,N_y)) $\wedge$ P.content$\neq$NULL

Send(T,proceed(P.content))
P.content $\leftarrow$ N_y

3. Accept content and reply to thread.

Receive(P,put(N_z))

P.content $\leftarrow$ N_z
Send(P.thread,proceed(P.content))
P.content $\leftarrow$ P.newcontent

4. Accept forward.

Receive(P,forward(P'))

P.forward $\leftarrow$ P'

5. Forward content.

P.forward$\neq$NULL $\wedge$ P.content$\neq$NULL

Send(P.forward,put(P.content))
P.forward $\leftarrow$ NULL
P.content $\leftarrow$ NULL
P.state $\leftarrow$ FREE

6. Serialize content requests (at manager).

Receive(M,get(P))

Send(M.tail,forward(P))
M.tail $\leftarrow$ P

Figure 7.12: Mobile state protocol: Migration of the content-edge.

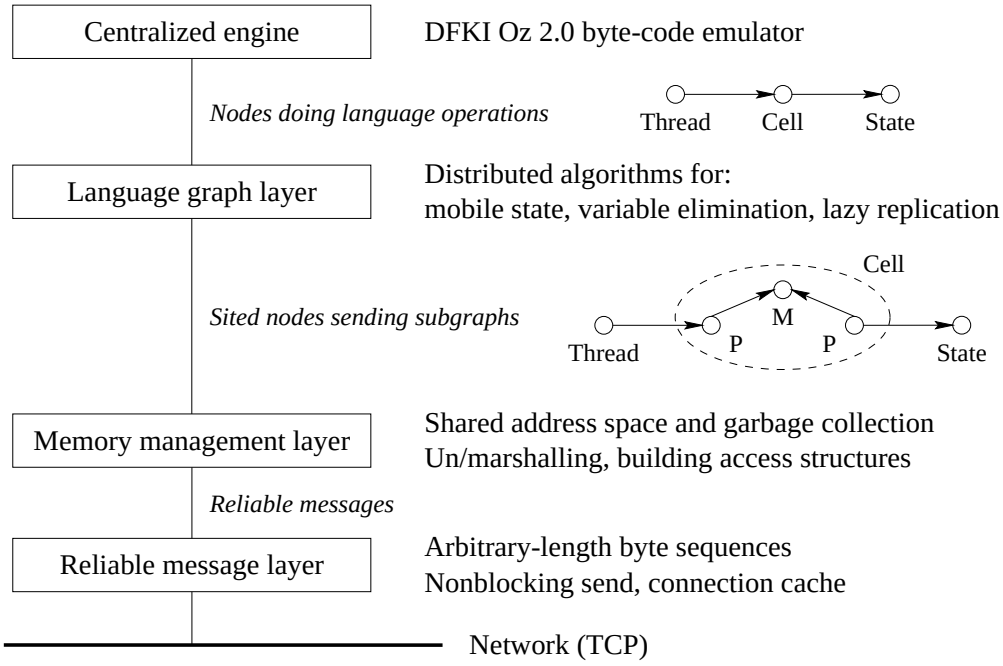


Figure 7.13: System architecture on one site.

is built. This section summarizes these aspects in sufficient detail to situate the protocol. Figure 7.13 shows an architecture for the execution of the distribution graph. This architecture is fully implemented and is being used for application development. Sections 7.8.1, 7.8.2, and 7.8.3 summarize the language graph layer, the memory management layer, and the reliable message layer.

Distribution is added as a conservative extension to a *centralized engine*. The extension is designed not to affect the centralized performance. The centralized engine executes internally all operations on local nodes. It is based on emulator technology and has similar or better performance than current Java emulators [60]. In particular, DFKI Oz 2.0 threads are much cheaper than Java threads. Operations on distributed entities are passed to the language graph layer. The *language graph layer* implements the distributed semantics for all Oz 2 entities, e.g., it decides when to do a local operation or a network communication. The language graph layer rests on a "bookkeeping" layer, the *memory management layer*. This layer implements the shared computation space, the building of access structures, and the distributed garbage collection. The *reliable message layer* implements transfer of arbitrary-length byte sequences between sites. It keeps a connection cache between sites and manages the message buffers. The *network* is the network interface of the host operating system, providing standard protocols such as TCP/IP.

7.8.1 Language Graph Layer

The distribution semantics of each Oz 2 entity is implemented in the language graph layer. Each entity has a separate protocol. There are three essentially different proto-

Item	Space (bytes)
Local object	36
Global object	
Active proxy	64
Passive proxy	44
Manager	44
Protocol messages	
get	15
forward	29
put	$15+S$

Table 7.4: Object Granularity in the Distributed Oz Implementation

cols: mobile state (cells, objects, and ports), variable elimination (logic variables), and lazy replication (procedures). Records and threads have trivial protocols. As illustrated in Figure 7.13 for cells, these protocols implement the illusion that all entities are centralized.

Table 7.4 gives the space usage of objects and messages in the mobile state protocol. Objects have a tag that defines them to be local, active proxy, passive proxy, or manager. This tag is combined with other fields to take up no extra space. The following extra run-time overhead is paid over a system that has only local objects. For each object operation, test the tag to see if the object is local or global, and if it is global, check if the content-edge is local. The local objects shown are of minimum size; add 4 bytes for each attribute and method. The global object sizes include the overhead for distributed garbage collection (see Section 7.8.2). A *passive* proxy is one that has not been called since the most recent local garbage collection. Other proxies are *active*. In the **put** message, S denotes the marshalled size of the state.

7.8.2 Memory Management Layer

Shared Computation Space

A two-level addressing scheme, using local and global addresses, is used to refer to nodes. The translation between local and global addresses is done automatically to maintain the following invariants. Nodes on the same site are always referred to by local addresses. Nodes on remote sites are always referred to by global addresses. Global addresses never change, since nodes in the distribution graph never change sites. If a node is remotely referenced, then it must have a global address. A node is remotely referenced if and only if it is referenced from another site or from a message in transit. If the node is no longer remotely referenced, then its global address will be reclaimed.

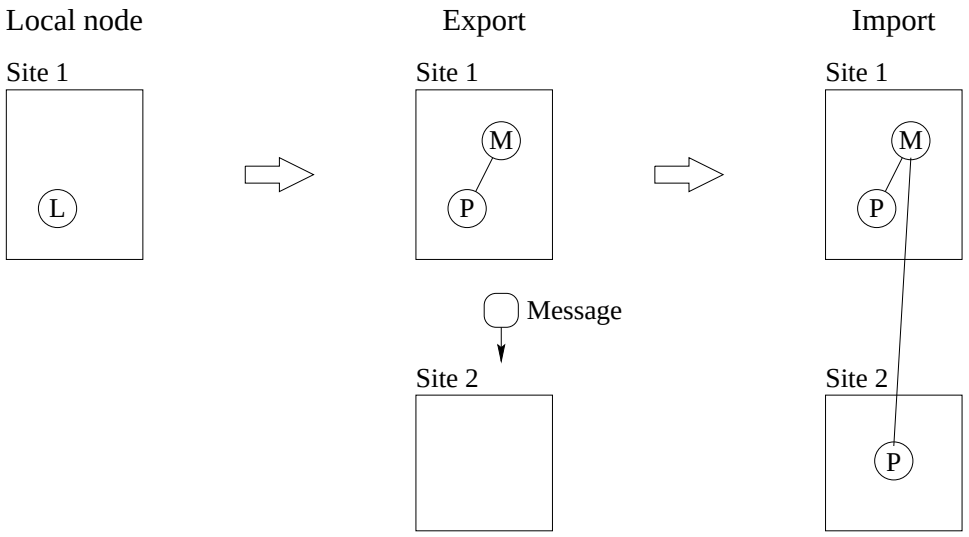


Figure 7.14: Globalizing a local node.

Building Access Structures

Access structures are built and managed automatically when language entities become remotely referenced. This happens whenever messages exchanged between nodes on different sites contain references to other nodes. If the reference is to a local node, then the memory management layer converts the local node into an access structure. We say the local node is *globalized* (see Figure 7.14). While the message is in the network, the access structure consists of a manager and one proxy. When the message arrives at the destination site, then a new proxy is created there. Access structures can reduce in size and disappear completely through garbage collection.

Distributed Garbage Collection

Distributed garbage collection is implemented by two cooperating mechanisms: a local garbage collector per site and a distributed credit mechanism to reclaim global addresses. Local collectors can be invoked at any time independently of other sites. The roots of local garbage collection are all nodes on the collector's site that are reachable from non-suspended thread nodes or are remotely referenced.

A global address is reclaimed when the node that it refers to is no longer remotely referenced. This is done by the credit mechanism, which is informed by the local garbage collectors. This scheme recovers all garbage except for cross-site cycles. The only cross-site cycles in our system occur between different objects or cells. Since records and procedures are both replicated, cycles between them will be localized to single sites. The credit mechanism does not suffer from the memory or network inefficiencies of previous reference-counting schemes [99].

We summarize briefly the basic ideas of the credit mechanism. Each global address is created with an integer (its *debt*) representing the number of *credits* that have been

given out to other sites and to messages. Any site or message that contains the global address must have at least one credit for the global address. The creation site is called the *owner*. All other sites are called *borrowers*. A node is remotely referenced if and only if its debt is nonzero.

Initially there are no borrowers, so the owner's debt is zero. The owner lends credits to any site or message that refers to the node and increments its debt each time by the number of credits lent. When a message arrives at a borrower, its credits are added to the credits already present. When a message arrives at the owner, its credits are deducted from the owner's debt. When a borrower no longer locally references a node then all its credits are sent back to the owner. This is done by the local garbage collector. When the owner's debt is zero then the node is only locally referenced, so its global address will be reclaimed.

Consider the case of a cell access structure. The manager site is the owner, and all other sites with cell proxies are borrowers. A proxy disappears when no longer locally referenced. It then sends its credit back to the manager. If the proxy contains the content-edge, then the content-edge is transferred back to the manager site as well. Remark that this removes a cross-site cycle within the cell access structure. When the manager recovers all its credit then it disappears, and the cell becomes a local cell again. When the local cell has no local references, then it is reclaimed. If the local cell becomes global again (because a message referring to it is sent across the network), then a new manager is created, completely unrelated to the reclaimed one.

7.8.3 Reliable Message Layer

The reliable message layer is the part of the Distributed Oz architecture that is closest to the operating system. This layer assumes a reliable transport protocol with no bounds on transfer time. For all entities except threads and ports, no assumptions are made on relative ordering of messages (no FIFO). For threads and ports, FIFO connections are made. The current prototype implements a cache of TCP connections to provide reliable transfer between arbitrary sites on a wide-area network [27]. Recent implementations of TCP can outperform UDP [20]. To send arbitrary-length messages from fair concurrent threads, the implementation manages its own buffers and uses nonblocking send and receive system calls. If some global addresses in a message require additional credit, then the message is put in a pending queue until all credit arrives.

7.9 Related Work

All systems that we know of except Emerald [72] and Obliq [21] do distributed execution by adding a distribution layer on top of a centralized language, e.g., CORBA [30, 96], DCE [122], Erlang [138], Java [120], Facile [80], and Telescript [89]. This has the disadvantage that distribution is not a seamless extension to the language, and

therefore distributed extensions to language operations (such as mobile objects or replicated data) must be handled by explicit programmer effort.

A better environment for distributed programming can be obtained by looking carefully at the entities of the language and conservatively extending their behavior to a distributed setting. For example, the Remote Procedure Call (RPC) [17] is designed to mimic centralized procedure calling and is therefore a precursor to the design given in this article. Following this integrated approach has two consequences. First, in order to carry it out successfully, the language must have a well-defined operational semantics that clearly identifies the entities. Second, to do the extensions right one must design a distributed algorithm for each entity. In the following sections, we take a closer look at distributed shared memory, Emerald, and Obliq.

7.9.1 Distributed Shared Memory

Distributed shared memory (DSM) [29, 122] has the potential to provide an adequate substrate for distributed programming. DSM has traditionally been viewed as a substrate for parallel programming, but work has been done on using it for distributed programming. We limit the discussion to distribution in software DSM. To achieve predictable network awareness, a language for distributed programming must be well-matched with its DSM layer. Following [29], we distinguish between page-based and library-based DSM.

Page-based DSM does not provide predictable network awareness. The units of distribution ("pages") do not correspond directly to language entities. This is too coarse grained for the applications we have in mind. It leads to false sharing. Munin [24], while page-based, provides programmer annotations for network awareness. A data item in memory can be annotated as read-only, migratory, write-shared, and so forth.

Library-based DSM provides sharing that is designed for particular data abstractions and hence can avoid the problems of granularity and false sharing. For example, Orca [14] provides network pointers, called "general graphs," and shared objects. Orca is designed for parallel applications. Linda [23] provides operations to insert and extract immutable data items, called "tuples," from a shared space. This is called a *coordination model*, and it can be added to any existing language. Linda does not address the language issues of network transparency.

Distributed Oz follows the library-based approach. The shared computation space is a DSM layer that is designed to support all language entities. The layer is extended to provide functionality that is not part of traditional DSMs. First, it supports single-assignment data in a strong form (logic variables) as well as other sharing protocols such as read-only data (values) and migratory data (objects). Second, as was briefly mentioned in Section 7.3, the system is open: sites can connect and disconnect dynamically. Although not impossible, we do not know of any DSM system that possesses this property. Third, the system is portable across a wide range of operating systems and processors. Fourth, the system can be extended to support precise failure detection.

7.9.2 Emerald

Emerald is a statically typed concurrent object-based language that provides fine-grained object mobility [71, 72]. The object system of Emerald is interesting in its own right. We limit the discussion to issues related to distribution. Emerald has distributed lexical scoping and is implemented efficiently with a two-level addressing scheme. Emerald is not an open system. Objects can be mutable or immutable. Objects are stationary by default and explicit primitive operations exist to move them. Having an object reference gives both the right to call and to move the object; these rights are separated in Distributed Oz. Immutable objects are copied when moved. Apart from object mobility, Emerald does not provide any special support for latency tolerance. There is no syntactic support for using objects as caches.

Moving a mutable object in Emerald is an atomic operation that clones the object on the destination site and aliases the original object to it. The result is that messages to the original object are passed to the new object through an aliasing indirection. If the object is migrated again, there will be two indirections, and so forth. The result is an aliasing chain. This chain is lazily shortened in two ways. First, if the object returns to a previously visited site, then the chain is short-circuited. Second, all message replies inform the message sender of the object's new site. If the object is lost because a site failure induces a break in the aliasing chain, then a broadcast is used to find the object again. Using broadcast does not scale up to many sites. As in Distributed Oz, failure is detected for single objects.

Because of the aliasing chain and possible broadcasting, it is difficult or impossible to predict the network behavior in Emerald or to guarantee that an object is independent of third-party sites. These problems are solved in Distributed Oz by using a manager node that is known to all proxies (see Section 7.7.2). This gives an upper bound of three on the number of network hops to get the object and guarantees that all third-party dependencies except for the manager site eventually disappear. Furthermore, the lack of an aliasing chain means that losing an object is so infrequent that it is considered as an object failure. There is therefore no need for a broadcast algorithm.

The Emerald system implements a distributed mark-and-sweep garbage collector. This algorithm is able to collect cross-site cycles, but it is significantly more complex than the Distributed Oz credit mechanism. It requires global synchronization, and it is not clear whether this scales up. It handles temporary network failures, but it is not clear how it behaves in the case of site failures.

7.9.3 Obliq

With Obliq, Distributed Oz shares the notions of dynamic typing, concurrency, state awareness, and higher-orderness with distributed lexical scoping. We differ from Obliq in two major ways: mobility control is a basic part of the design, and logic variables introduce a fundamental dataflow element. Another difference is that Distributed Oz is object-oriented with a rich concurrent object system, while Obliq is object-based. While it is outside the scope of this article, we mention that Oz 2 is a powerful con-

straint language that is being used in problem-solving research. The constraint aspects of Oz 2 are orthogonal to the distribution aspects given in this article.

Obliq has taken a first step toward the goal of conservatively extending language entities to a distributed setting. Obliq distinguishes between *values* and *locations*. Moving values causes them to be copied (replicated) between sites. Moving locations causes network references to them to be created.

Distributed Oz takes this approach for the complete language, consisting of seven language entities. Each of these entities has a distributed algorithm that is used to remotely perform an operation on the entity (see Figure 7.3). The algorithms are designed to preserve the language semantics while providing a simple model for the communication patterns.

It is interesting to compare object migration in Obliq with Distributed Oz mobile objects. Obliq objects are stationary. Object migration in Obliq can be implemented in two phases by cloning the object on another site and by aliasing the original object to the clone. These two phases must be executed atomically to preserve the integrity of the object's state. According to Obliq semantics, the object must therefore be declared as *serialized*. To be able to migrate these objects, the migration procedure must be executed internally by the object itself (be *self-inflicted*, in Obliq terminology). The result is an aliasing chain.

Oz 2 objects are defined as procedures that have access to a cell. The content-edge refers to the current object state. Mobility is obtained by making the cell mobile. When a method is invoked on a remote site, the content-edge is first moved to that site. Integrity of the state is preserved because the cell continues to obey the language semantics. The implementation uses a distributed algorithm to move the content-edge (see Section 7.7). Because mobility is part of a cell's distributed semantics, there are no chains of indirections. This is true as well for mostly stationary objects, which use the port protocol (see Section 7.7.4).

7.10 Conclusions, Status, and Current Work

We have presented the design of a language for distributed programming, Distributed Oz, in which the concept of *mobility control* has been incorporated in a fundamental way. We define the language in terms of *two* operational semantics, namely a language semantics and a distributed semantics. The language semantics ignores the notion of site and allows reasoning about correctness and termination. The distributed semantics gives the network behavior and allows the writing of programs that use the network predictably and efficiently. Mobility control is part of the distributed semantics.

The main contribution of this article is a system for network-transparent distribution in which the programmer's control over network communication patterns is both predictable and easy to understand. To make object migration predictable, the implementation uses a distributed algorithm to avoid forwarding chains through intermediate sites. This guarantees that all dependencies to third-party sites except for the manager site eventually disappear.

We show by example how this approach can simplify the task of distributed programming. We have designed and implemented a prototype shared graphic editor that is efficient on networks with high latency yet is written in a fully network-transparent manner. We give Oz 2 code that shows how the mobility of objects can be precisely controlled and how this control can be added independently of the object's definition.

We give a formal definition of the mobile state protocol, and we prove that it implements the language semantics. This implies that the implementation of cells, objects, and ports in Distributed Oz is network transparent.

We outline the system architecture including the distributed memory management and garbage collection. A prototype implementation of Distributed Oz exists that incorporates all of the ideas presented in this article. The prototype maintains the semantics of the Oz 2 language. The implementation is an extension of the centralized DFKI Oz 2.0 system and has been developed jointly by the German Research Center for Artificial Intelligence (DFKI) [118] and the Swedish Institute of Computer Science (SICS). DFKI Oz 2.0 is publicly available[17] and has a full-featured development environment.

This article has presented one part of the Distributed Oz project. This work is being extended in several ways. An important unresolved issue is to find high-level abstractions for fault tolerance that separate the application's functionality from its fault-tolerant behavior. Providing the basic primitives, e.g., precise failure detection, is not difficult. Other current work includes improving the efficiency and robustness of the prototype, using it in actual applications, improving the support for open computing, and building the standard services needed for distributed application development. Future work includes adding support for resource management and multiprocessor execution (through "virtual sites"), and adding security. The main research question to be addressed is how to integrate these abilities without compromising network transparency.

7.11 APPENDIX

7.11.1 Correctness Proof of the Mobile State Protocol

This appendix gives a proof that the mobile state protocol as defined in Section 7.7.4 implements the language semantics of the exchange operation as defined in Section 7.7.1. We have given three specifications of the exchange operation:

(LS) Language reduction rule (Section 7.7.1)
(DS) Distributed reduction rule (Section 7.7.1)
(MP) Mobile state protocol (Section 7.7.4)

Theorem T1 (Section 7.7.1) implies that (DS) implements (LS). It remains to be shown that (MP) implements (DS). We do this in two parts. First, in Section 7.11.2 we prove safety and liveness of the mobile state protocol. Namely, all reachable configurations

[17] At `http://www.ps.uni-sb.de`

satisfy the *chain invariant* defined in Section 7.11.3, and a request for the content-edge on a node that does not have it causes it to arrive exactly once. Then, in Section 7.11.6 we use these results first to prove that the mobile state protocol is observationally equivalent to a much simpler protocol. We then show that the latter implements the distributed reduction rule for exchange.

7.11.2 Mobile State Protocol Correctly Migrates the Content-edge

This section proves that the mobile state protocol correctly implements the migration of the content-edge. We follow the definitions and notations introduced in Section 7.7. The proof is structured around a global distributed data structure that we call a *chain*, which is defined in Section 7.11.3 by an invariant of the distribution graph [85, 124]. Section 7.11.4 proves that the mobile state protocol satisfies the chain invariant. Section 7.11.5 proves that requesting the content causes it to arrive exactly once.

Informally, a chain consists, at any given instant, of the known path of the content among the proxy nodes (see Figure 7.15). That is, it is a sequence of proxy nodes such that the first node contains the content or will eventually receive it and that all nodes will eventually pass the content on to the next node in the chain. The chain grows if new content-edge requests are more frequent than the rate at which the content is forwarded among proxies. If there are no new requests, then the chain shrinks to length one. The chain is defined formally as part of the proof. Reasoning in terms of the chain makes proving properties of the protocol tractable.

For the proof, we ignore the attributes thread and newcontent and the operations concerning them. We are interested only in whether the content attribute is NULL or non-NULL. We use the special value CVAL to represent any non-NULL value. We assume the initial P_1.content and the N_y argument in request messages are both CVAL. We show that there is only one node or message in the network that contains the value CVAL. Any node may request the content, and we show that the protocol guarantees that the content will eventually arrive exactly once. After arriving at the node, the content will eventually leave if another node requests it.

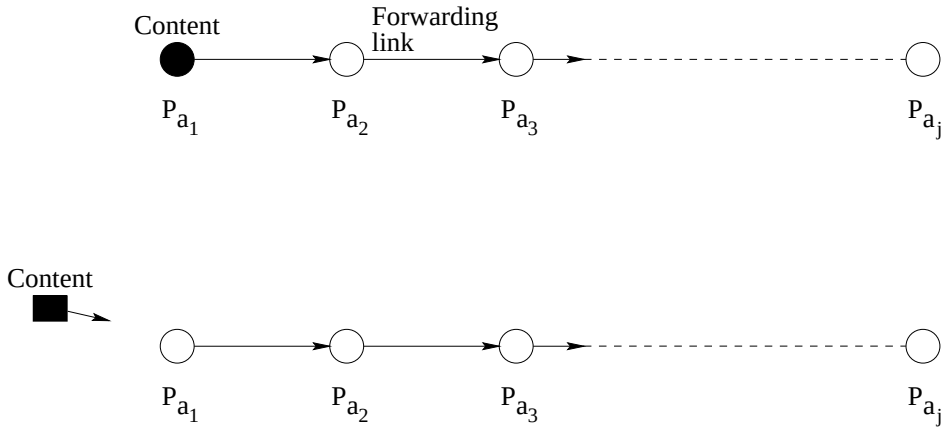


Figure 7.15: Two forms of a chain.

7.11.3 Chain Invariant

The chain invariant I is defined as follows on the distribution graph:

$$
\begin{aligned}
I &= I_p \wedge I_a \wedge I_b \wedge I_c \wedge I_t \wedge I_u && (7.1)\\
I_p &= A, B, C \text{ form a partition of } \{1, ..., k\} && (7.2)\\
I_a &= A = \{a_1, ..., a_j\} \ \wedge \ j > 0 \ \wedge \\
&\quad \left(1 \leq \forall i < j : \left\{ \begin{array}{l} P_{a_i}.\text{state} = \text{CHAIN} \\ P_{a_i}.\text{forward} \in \{\text{NULL}, a_{i+1}\} \\ P_{a_i}.\text{forward} = a_{i+1} \ \oplus \ a_i : \text{forward}(a_{i+1}) \in N \end{array} \right. \right) \ \wedge \\
&\quad P_{a_j}.\text{state} = \text{CHAIN} \ \wedge \ P_{a_j}.\text{forward} = \text{NULL} \ \wedge \ M.\text{tail} = a_j && (7.3)\\
I_b &= \forall i \in B : \left(\begin{array}{l} P_i.\text{state} = \text{CHAIN} \ \wedge \ P_i.\text{forward} = \text{NULL} \ \wedge \\ M : \text{get}(i) \in N \end{array} \right) && (7.4)\\
I_c &= \forall i \in C : (P_i.\text{state} = \text{FREE} \ \wedge \ P_i.\text{forward} = \text{NULL}) && (7.5)\\
I_t &= P_{a_1}.\text{content} \in \{\text{NULL}, \text{CVAL}\} \ \wedge \\
&\quad (P_{a_1}.\text{content} = \text{CVAL} \ \oplus \ a_1 : \text{put}(\text{CVAL}) \in N) \ \wedge \\
&\quad 1 \leq \forall i \leq k, i \neq a_1 : P_i.\text{content} = \text{NULL} && (7.6)\\
I_u &= \text{All messages in } N \text{ are unique and explicitly mentioned in } I && (7.7)
\end{aligned}
$$

Informally, I_a states that all the proxy nodes in A form a chain, where $\oplus$ denotes the exclusive-or (see Figure 7.15). We call P_{a_1} the *head* of the chain and P_{a_j} the *tail* of the chain. I_b states that all the proxy nodes in B will eventually become part of the chain. When it is known from which source node a target node will receive the content, then the target node is considered part of the chain. I_c states that all other proxy nodes (which are in C) are not part of the chain. The content invariant I_t states that there is exactly one content-edge in the system and that it belongs to the head of the chain. Writing the uniqueness invariant I_u as an explicit formula is left as an exercise for the reader.

Figure 7.16 illustrates the partitioning of the proxy nodes in classes C, B, and A.

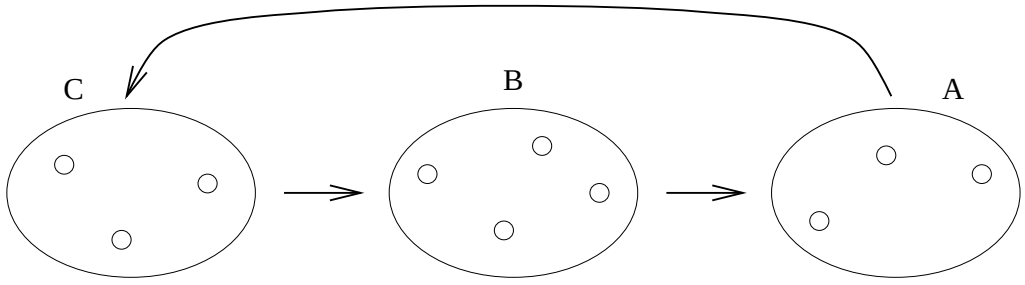


Figure 7.16: Classifying the proxy nodes.

During execution, a proxy node changes class according to the arrows. The node is in class C when it does not have the content-edge and has not requested it. The node moves from class C to B when the content-edge is requested. The node moves from class B to A when the manager is informed of the request. The node moves from class A to C when the content-edge leaves the node. Within class A, the content-edge moves from one node to the next in the chain. The node reaches the head of the chain when it receives the content-edge.

7.11.4 Safety Theorem

Theorem 2 *(S).* The chain invariant (formula I in Section 7.11.3) is an invariant of the mobile state protocol (Section 7.7.4).

Proof. It is clear that I holds in the configuration C_0 with $N = \emptyset$, $A = \{1\}$, $B = \emptyset$, and $C = \{2, ..., k\}$. Consider a configuration C_i in which I holds. Then we show that I also holds in configuration C_{i+1}:

1. Rule 1 is only applicable when $C \neq \emptyset$. Applying rule 1 removes one element from C and adds it to B. This affects I_b and I_c. It is clear that both formulas and I_u continue to hold.

2. Rule 2 is applicable when P.content=CVAL. Since the request message's second argument is CVAL, therefore applying rule 2 changes nothing in the invariant.

3. Rule 3 is applicable when the second alternative $(a_1 : \text{put(CVAL)} \in N)$ of the disjunction in I_t holds. Applying the rule causes only the first alternative to hold, which maintains the truth of I_t and I_u.

4. Rule 4 is applicable when the second alternative $(a_i : \text{forward}(a_{i+1}) \in N)$ of a disjunction in I_a holds. Applying the rule causes only the first alternative to hold, which maintains the truth of I_a and I_u.

5. Rule 5 is applicable when the first alternatives of the disjunctions in I_a and I_t hold. From I_a and I_t we deduce that when rule 5 is applicable, it is applicable at node P_{a_1}, that P_{a_1}.forward $= a_2$, and that therefore $j \geq 2$. Applying the rule removes a_1 from A and adds it to C, maintaining the truth of I_a and I_c. Since a_2 : put(CVAL) is added to the network, the truth of I_t is maintained.

6. Rule 6 is applicable when B is nonempty. Applying the rule removes one element from B and adds it to A. Reasoning from the value of M.tail shows that this element becomes the new a_j. In other words, each reduction of the manager node adds one element to the chain. The truth of I_u is maintained.

This proves the theorem. □

7.11.5 Liveness Theorem

Theorem 3 *(L).* Given the fairness assumptions of Section 7.7.4, requesting the content at a proxy node will cause it eventually to arrive once.

Proof. The statement of the theorem means that for all proxy nodes P we have one of the following three cases:

1. P.content=CVAL. The content is at the node.

2. P.content=NULL $\wedge$ P.state=FREE. Reducing rule 1 causes P.content=CVAL eventually to become true once through the application of rule 3.

3. P.content=NULL $\wedge$ P.state=CHAIN. Then P.content=CVAL will eventually become true once through the application of rule 3.

Case 1 is evident. In case 2, rule 1 is clearly applicable, and reducing it gives the condition of case 3. It is clear that the only way in which P.content=CVAL can become true is through the reduction of rule 3. In what follows, we consider only case 3.

In case 3, rule 1 has been reduced once at node P. This causes an eventual reduction of rule 6 once, which results in a configuration C_x with invariant I_x such that P's index is in A_x. If $|A_x| = 1$ then the proof is done. Let us assume without loss of generality that $|A_x| \geq 2$. What is the relationship between A_x and A_{x+1}? That is, what happens to A_x during reduction of a single rule? Inspecting the rules shows that there are three possibilities (assume $|A_x| = j$):

$$A_{x+1} = A_x \tag{7.1}$$
$$A_{x+1} = A_x \setminus \{a_1\} \tag{7.2}$$
$$A_{x+1} = A_x \cup \{a_{j+1}\} \tag{7.3}$$

We must show that in any configuration C_x with $|A_x| \geq 2$, possibility (2) is eventually executed. From inspection, this is only possible by applying rule 5.

We show that from C_x it will always become possible to apply rule 5. We use the fact that when rules 3, 4, or 5 become applicable, they stay applicable until reduced.

Assume that I_a contains a_1 : forward(a_2). Then we can apply rule 4. Therefore we can assume without loss of generality that P_{a_1}.forward $= a_2$. Similarly, by possibly applying rule 3, we can assume P_{a_1}.content = CVAL. The equations P_{a_1}.forward $= a_2$ and P_{a_1}.content = CVAL imply the conditions of rule 5. Therefore rule 5 will eventually be reduced. This proves that the head a_1 is eventually removed from A_x. Since new nodes are only added at the tail, this proves that all elements will eventually receive the token. This shows that the content eventually arrives. We know that nodes are only removed at the head and that the chain does not contain cycles. This shows that the content arrives exactly once. □

7.11.6 Mobile State Protocol Implements Distributed Semantics

The proof is presented as two lemmas and a theorem. From the two lemmas it follows that the mobile state protocol is observationally equivalent to a much-simplified protocol with one proxy node, one reduction rule, and no manager. In a similar manner to Section 7.7.1, we then use abstraction to show that the behavior of this simplified protocol exactly corresponds to the distributed semantics of exchange.

Lemma 1 *(A).* Invoking **Send**(P,request(T,N)) will eventually result in exactly one atomic execution at P of the derived rule:

$$\text{(EXCH)}\quad \frac{\textbf{Receive}(\text{P,request(T,N)}) \ \wedge\ \text{P.content}\neq\text{NULL}}{\begin{array}{l}\textbf{Send}(\text{T,proceed(P.content)})\\ \text{P.content} \leftarrow \text{N}\end{array}}$$

Proof. We prove the result in two parts. We first show that eventually rule 1 or rule 2 is reduced exactly once. Assume the **Receive**(P,request(T,N)) condition corresponding to the send becomes true at proxy P. Enumerating all possible values of P.state and P.content gives the following four cases:

1. P.state=FREE $\wedge$ P.content$\neq$NULL. We show that this condition never occurs in a reachable configuration. By Theorem S, the chain invariant I is valid in all reachable configurations. Then according to I_t, P.content$\neq$NULL means that P=P_{a_1}. From I_a we see that P_{a_1}.state=CHAIN.

2. P.state=FREE $\wedge$ P.content=NULL. Rule 1 is applicable and stays applicable, so by the fairness assumption eventually it is reduced.

3. P.state=CHAIN $\wedge$ P.content$\neq$NULL. Rule 2 is applicable. Either it is eventually reduced, in which case all is well, or it is never reduced. The latter can only happen if rule 5 is reduced, which makes P.state=FREE $\wedge$ P.content=NULL. The previous case then applies.

4. P.state=CHAIN $\wedge$ P.content=NULL. No rule is applicable. However, since P.state=CHAIN, this means that rule 1 has been reduced by the reception of another request message. By Theorem L, eventually P.content$\neq$NULL. When this happens, the previous case applies.

Distributed Semantics	Distribution Graph Notation
Representatives $\mathtt{X}_q$, $\mathtt{Y}_q$, $\mathtt{Z}_q$	Nodes $\mathrm{N_X}$, $\mathrm{N_Y}$, $\mathrm{N_Z}$ on site q
Cell representative $\mathtt{C}_q$	Cell proxy P_q
$\mathtt{C}_1\mathtt{=n} \wedge ... \wedge \mathtt{C}_k\mathtt{=n}$	Access structure containing P_i for $1 \leq \forall i \leq k$
$(\mathtt{n:Z})_p \wedge 1 \leq \forall i \leq k, i \neq p : (\mathtt{n:}\bot)_i$	P_p.content=$\mathrm{N_Z}$, $1 \leq \forall i \leq k, i \neq p$: P_i.content=NULL
$\mathtt{\{Exchange\ C\ X\ Y\}}_q$	exchange_and_bind(P,$\mathrm{N_X}$,$\mathrm{N_Y}$) in thread on site q, until just before bind operation
$(\mathtt{X=Z})_q$	bind($\mathrm{N_X}$,$\mathrm{N_Z}$) in thread on site q

Table 7.5: Correspondence between distributed semantics and graph notation

This shows that eventually rule 1 or rule 2 will be reduced. The first condition **Receive**(P,request(T,N)) of the (EXCH) rule is therefore taken care of. We now show that in each case the lemma is true:

1. Rule 1 is reduced. By Theorem L, the content arrives exactly once by the reduction of rule 3. This makes true the second condition P.content≠NULL of the (EXCH) rule. Between the reduction of rule 1 and the arrival of the content, only rule 4 is potentially applicable. Reducing rule 4 is irrelevant since it only changes the value of P.forward, which does not change the value of any other node attribute nor the applicability of any rules. When the content arrives, rule 3 becomes applicable. Reduction of rule 3 makes the lemma true.

2. Rule 2 is reduced. Inspection of rule 2 makes it clear that the lemma is true.

This proves the lemma. □

Lemma 2 *(B).* The P.content value at the end of one (EXCH) rule is used as the P.content value at the beginning of exactly one other (EXCH) rule, or of no (EXCH) rules if no further (EXCH) rules are executed.

Proof. By Theorem S, either P.content≠NULL on exactly one proxy or there is exactly one put(N) message in transit. We also know that the transfer of P.content between two proxies conserves its value, since the transfer can only be done by reducing rule 5 at the source and rule 3 at the destination. This proves the lemma. □

Theorem 4 *(T2).* The mobile state protocol of Section 7.7.4 implements the distributed semantics of Section 7.7.1.

Proof. We first make clear the correspondence between the notations of Sections 7.7.1 and 7.7.4 (Table 7.5). Then we show that the execution of the exchange procedure follows exactly the distributed reduction rule.

By Lemma A, executing exchange_and_bind(P,$\mathrm{N_X}$,$\mathrm{N_Y}$) in thread T on site q eventually reduces one (EXCH) rule. The bind operation becomes applicable when the message sent by this rule is received by the thread. The receive therefore marks the

end of the exchange reduction. Denote P.content=N_Z just before entering the (EXCH) rule's body. By Lemma B, N_Z is the result of the previous exchange. Then, assuming the correspondence in Table 7.5 holds just before the (EXCH) rule, we show that the correspondence holds just after the (EXCH) rule:

1. The nodes of the access structure are undisturbed.
2. P_q.content=N_Y and by Theorem S, $1 \leq \forall i \leq k, i \neq q$: P_i.content=NULL.
3. The operation bind(N_X,N_Z) is applicable in thread T after the exchange.

This corresponds exactly to the distributed reduction rule of Section 7.7.1. □

Acknowledgements

We thank Luc Onana for fruitful discussions that led to mobile ports and to the proof given in Appendix 7.11.2. We thank Christian Schulte for his help with the shared editor. We thank Michel Sintzoff, Joachim Niehren, and the anonymous referees for their perceptive comments that permitted us to improve and complete the presentation.

Chapter 8

Using logic variables in distributed computing

Efficient Logic Variables for Distributed Computing

SEIF HARIDI
Swedish Institute of Computer Science (SICS)

PETER VAN ROY
Université Catholique de Louvain and SICS

PER BRAND
Swedish Institute of Computer Science

MICHAEL MEHL and RALF SCHEIDHAUER
German Research Center For Artificial Intelligence (DFKI)
and
GERT SMOLKA
Universität des Saarlandes and DFKI

8.1 Abstract

We define a practical algorithm for distributed rational tree unification and prove its total correctness in both the off-line and on-line cases. We derive the distributed algorithm from a centralized one, showing clearly the trade-offs between local and distributed execution. The algorithm is used to realize logic variables in the Mozart system, which implements the Oz language. Oz appears to the programmer as a concurrent object-oriented language with dataflow synchronization. Oz is defined in terms of a general execution model that consists of concurrent constraints extended with explicit state and higher-orderness. We show that logic variables can easily be added to the more restricted models of Java and ML. We present common distributed programming idioms in a network-transparent way using logic variables. We show that in common cases the algorithm maintains the same message latency as explicit message passing. In addition, it is able to handle uncommon cases that arise from the properties of latency tolerance and third-party independence. This is evidence that using logic variables in distributed computing is beneficial at both the system and language levels. At the system level, they improve latency tolerance and third-party independence. At the language level, they help make network-transparent distribution practical.

8.2 Introduction

Logic variables were first studied in the context of logic programming [105, 136]. They remain an essential part of logic programming and constraint programming systems [127, 67]. In the context of the Distributed Oz project, we have come to realize their usefulness to distribution [55, 118]. Logic variables express dependencies between computations without imposing an execution order. This property can be exploited in distributed computing:

- Two basic concerns in distributed computing are latency tolerance and third-party independence. We say a program is *third-party independent* if its execution is unaffected by sites that are not currently involved in the execution. We show that using logic variables instead of explicit message passing can reduce the effect of both concerns with little programming effort.

- With logic variables we can express common distributed programming idioms in a network-transparent manner that results in optimal or near-optimal message latency. That is, the same idiom that works well in a centralized setting also works well in a distributed setting.

The main contribution of this article is a practical distributed algorithm for rational tree unification that realizes these benefits. The algorithm is used to realize logic variables in the Mozart system. We formally define the algorithm and prove that it satisfies safety and liveness properties in both the off-line and on-line cases.

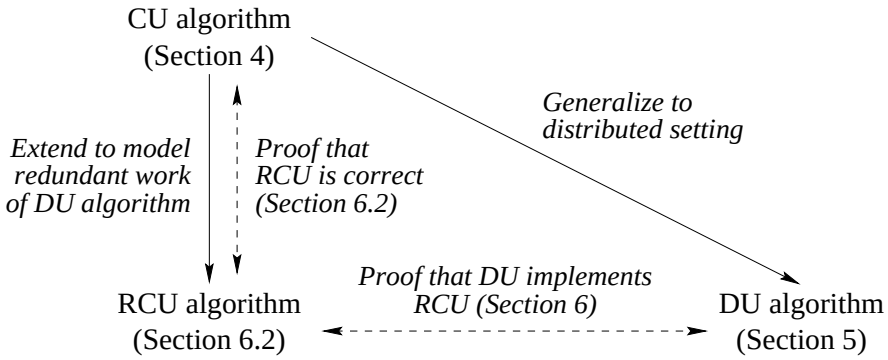


Figure 8.1: Defining the algorithm and proving it correct

From the programmer's point of view, the use of logic variables adds a dataflow component to program execution. In a first approximation, this component can be completely ignored. That is, it is invisible to the programmer whether or not a thread temporarily blocks while waiting for a variable's value to arrive. Programs can be developed using well-known techniques of concurrent object-oriented programming [79]. In a second approximation, the dataflow component greatly simplifies many concurrent programming tasks [51, 15].

This article consists of two parts that may be read independently of each other. The first part, Section 8.3, motivates and discusses in depth the use of logic variables in concurrent and distributed programming. Section 8.3.1 introduces a general execution model, its distributed extension, and the concept of logic variable. Section 8.3.2 gives the key ideas of the distributed unification algorithm. Section 8.3.3 shows how to express basic concepts in concurrent programming using logic variables. Section 8.3.4 expresses common distributed programming idioms in a network-transparent manner with logic variables. We show that the algorithm provides good network behavior for these examples. Finally, Section 8.3.5 shows how to add logic variables in an orthogonal way to Java and ML, taken as representative examples of object-oriented and functional languages.

The second part, Section 8.4 and following, defines the distributed unification algorithm, proves its total correctness (see Figure 8.1), and discusses its implementation. Section 8.4 defines the formal representation of logic variables and data structures. This section also defines configurations and executions and introduces the reduction rule notation used to define algorithms. Section 8.5 defines the CU algorithm, which implements off-line centralized unification, and summarizes well-known results about its correctness. By *off-line* we mean that the set of equations is finite and initially known. Section 8.6 defines the DU algorithm, which implements off-line distributed unification. Section 8.7 defines the RCU algorithm, which modifies the centralized algorithm to reflect the redundant work done by the DU algorithm. The section then proves that the DU algorithm is a correct implementation of the CU and RCU algo-

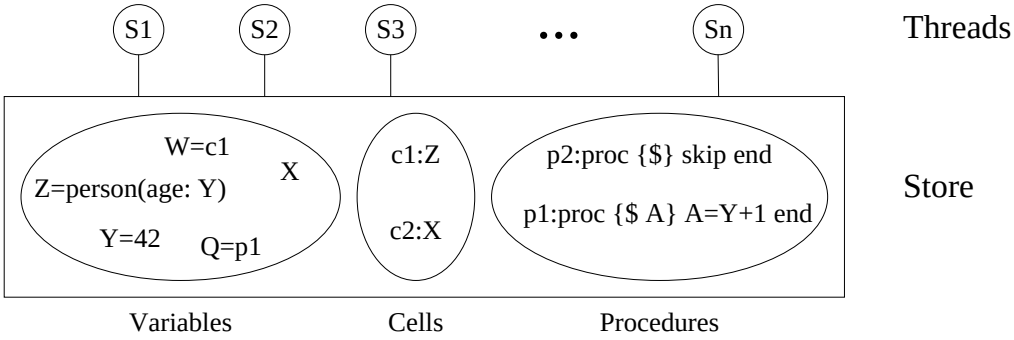


Figure 8.2: The Oz execution model

rithms. Section 8.8 defines on-line versions of the CU and DU algorithms. By *on-line* we mean that new equations can nondeterministically be introduced at any moment. We define the *finite size* property, and prove that given weak fairness, every introduced equation that satisfies this property is eventually entailed by the store for both algorithms. Section 8.9 defines the algorithm used in the Mozart system, which implements the on-line DU algorithm.

8.3 Logic variables in concurrent and distributed settings

This section motivates our unification algorithm by showing its usefulness to distributed programming. First Section 8.3.1 introduces our execution model and its notation. Then Section 8.3.2 gives the key ideas of the algorithm. This is followed by Section 8.3.3 which gives programming examples showing the usefulness of logic variables for basic tasks in concurrent programming. Section 8.3.4 continues with tasks in distributed programming. We explain in detail the network behavior of our algorithm for these tasks. Finally, Section 8.3.5 shows how to add logic variables to other languages including the Java and ML families.

8.3.1 Basic concepts and notation

As a framework for logic variables, we introduce a general execution model that can accommodate most programming languages. Underlying it is a formal model called *concurrent constraints* [116, 108] that contains logic variables as a basic ingredient. Some uses of logic variables, e.g., synchronization and communication, already appear in this model. The general execution model, called Oz execution model, extends the concurrent constraint model with explicit state and higher-orderness. Other uses of logic variables, e.g., locks, become possible when explicit state is added.

This section gives the essential characteristics of the Oz execution model and how it is distributed. Later on, we show how to add logic variables to the more restricted models of Java and ML. The advantage of using a general formal model is that it allows us to define precisely what the examples do. It is straightforward to compile Java or ML to Oz; the converse is not as easy.

The Oz execution model

The Oz execution model consists of dataflow threads observing a shared store (see Figure 8.2). Threads contain statement sequences S_i and communicate through shared references into the store. A thread is *dataflow* if it only executes its next statement when all the values the statement needs are available. If the statement needs a value that is not yet available, then the thread automatically blocks until it can access that value. We add the fairness condition that every thread that is not blocked eventually executes its next statement. As we shall see, data availability in the Oz model is implemented using logic variables.

The shared store is not physical memory, rather it is an abstract store that only allows legal operations for the entities involved, i.e., there is no direct way to inspect their internal representations. The store consists of three compartments, namely logic variables (with optional bindings), cells (named mutable pointers, i.e., explicit state), and procedures (named lexically-scoped closures, i.e., higher-orderness). Variables can reference the names of procedures and cells. Cells point to variables. The external references of threads and procedures are variables. When a variable is bound, it disappears, that is, all threads that reference it will automatically reference the binding instead. Variables can be bound to any entity, including other variables. The variable and procedure stores are monotonic, i.e., information can only be added to them, not changed or removed. Because of monotonicity, a thread that is not blocked is guaranteed to stay not blocked until it executes its next statement.

The Oz language

All Oz execution can be defined in terms of a kernel language whose semantics are outlined in [51, 132]. The current Oz language is called Oz 2 to distinguish it from an earlier language, Oz 1, whose kernel language is called Oz Programming Model (OPM) [116]. Oz 1 was designed for fine-grained concurrency and implicit exploitation of parallelism. Oz 2 abandons this model in favor of explicit control over concurrency by means of a thread creation construct. We do not discuss Oz 1 further in this paper.

Figure 8.3 defines the abstract syntax of a statement S in the Oz kernel language. Statement sequences are reduced sequentially inside a thread. All variables are logic variables, declared in an explicit scope defined by the **`local`** statement. Values (records, numbers, names, etc.) are introduced explicitly and can be equated to variables. A *name* is a unique unforgeable constant that has no external representation. A new name is created by calling `NewName`. Procedures are defined at run-time with the

S ::=	$S\ S$	Sequence
|	X=`f(`l_1`:`Y_1 `...` l_n`:`Y_n`)` |	Value
	X=`<number>` | X=`<atom>` | `{NewName` X`}`	
|	**`local`** X_1 `...` X_n **`in`** S **`end`** | X=Y	Variable
|	**`proc`** `{`X Y_1 `...` Y_n`}` S **`end`** | `{`X Y_1 `...` Y_n`}`	Procedure
|	`{NewCell` Y X`}` | `{Exchange` X Y Z`}` | `{Access` X Y`}`	State
|	**`if`** X **`then`** S **`else`** S **`end`**	Conditional
|	**`thread`** S **`end`** | `{GetThreadId` X`}`	Thread
|	**`try`** S **`catch`** X **`then`** S **`end`** | **`raise`** X **`end`**	Exception

Figure 8.3: The Oz kernel language

`proc` statement and referred to by a variable. Procedure applications block until the first argument references a procedure name. State is created explicitly by `NewCell`, which creates a *cell*, a mutable pointer into the variable store. Cells are updated by `Exchange` and read by `Access`. The **`if`** statement defines a conditional that blocks until its condition is true or false in the variable store. Threads are created explicitly with the **`thread`** statement. Each thread has a unique identifier that is used for thread-related operations. Exception handling is dynamically scoped. The **`try`** statement defines a scope and the **`raise`** statement raises an exception that is caught by the innermost enclosing scope.

The full Oz language is defined by transforming all its statements into this kernel language. Oz supports idioms such as objects, classes, reentrant locks, and a variety of channel called "ports" [116, 132]. The system implements them efficiently while respecting their definitions. We give a brief summary of each idiom's definition. For clarity, we have made small conceptual simplifications. Full definitions are given in [51].

- **Object**. An object is essentially a one-argument procedure `{Obj M}` that references a cell, which is hidden by lexical scoping. The cell holds the object's state. The argument `M` indexes into the method table. A method is a procedure that is given the message and the object state, and calculates the new state.

- **Class**. A class is essentially a record that contains the method table and attribute names. A class is defined through multiple inheritance, and any conflicts are resolved at definition time when building its method table.

- **Reentrant lock**. A reentrant lock is essentially a one-argument procedure `{Lck P}` used for explicit mutual exclusion, e.g., of method bodies in objects used concurrently. `P` is a zero-argument procedure defining a critical section. Reentrant means that the same thread is allowed to reenter the lock. Calls to the lock may therefore be nested. The lock is released automatically if the thread in the body terminates or raises an exception that escapes the lock body.

- **Port**. A port is an asynchronous channel that supports many-to-one communication. A port `P` encapsulates a stream `S`. A *stream* is a list with unbound tail. The operation `{Send P M}` adds `M` to the end of `S`. Successive sends from the same thread appear in the order they were sent.

The distribution model

The Mozart system implements Distributed Oz, which is a conservative extension to the centralized Oz language [34] that completely separates functionality from distribution structure. That is, Oz language semantics are unchanged[1] while adding predictable and programmable control over network communication patterns. Porting existing Oz programs to Distributed Oz requires essentially no effort.

Allowing a successful separation of functionality from distribution structure puts severe restrictions on a language. It would be almost impossible in C++ because of its complex, informal semantics and because the programmer has full access to all underlying representations [119]. It is possible in Oz because of the following properties:

- Oz has a simple formal foundation that does not sacrifice expressiveness or efficient implementation. Oz appears to the programmer as a concurrent object-oriented language whose basic functionality is comparable to modern languages such as Java. The current emulator-based implementation is competitive with Java emulators [59, 58]. Standard techniques for concurrent object-oriented design apply to Oz [79]. Furthermore, Oz introduces powerful new techniques that are not supported by Java [51]. Some of these techniques are presented here.

- Oz is both a state-aware and dataflow language. That is, language entities can be classified naturally into stateless, single assignment, and stateful. This helps give the programmer control over network communication patterns in a natural manner. Stateless data includes procedures and values, which can safely be copied to many sites [7]. Stateful data includes objects, which at any instant must reside on just one site [132]. Single assignment data includes logic variables, whose dataflow synchronization allows to decouple calculating a value from sending it across the network.

- Oz is a fully dynamic and compositional language. That is, Oz is dynamically typed and all entities are first-class. By *dynamically typed* we mean that its type structure is checked at run-time. This makes it easy to implement fully open distribution, in which independent computations can connect and disconnect at will. When connected they can communicate as if they were in the same centralized process. For example, it is possible to define a class `C` in one computation, pass `C` to an *independent* computation that has never before heard of `C`, let the independent computation define a class `D` inheriting from `C`, and pass `D` back to the original computation [130, 54].

[1] Only ports are changed slightly to better model asynchronous FIFO communication between sites [132].

- Oz provides language security. That is, all references to language entities are created and passed explicitly. An application cannot forge references nor access references that have not been explicitly given to it. The underlying representation of language entities is inaccessible to the programmer. This is a consequence of the abstract store and a kernel language with lexical scoping and first-class procedures. These are essential properties to implement a capability-based security policy, which is important in open distribution.

The Distributed Oz execution model extends the Oz execution model by giving a distributed semantics to each language entity. The distributed semantics defines the network behavior when language entities are shared between sites. The semantics is chosen carefully to give predictable control over network communication patterns. The centralized semantics is unchanged: we say the model is *network-transparent* [21]. In the current system, language entities are put in four categories. Each category is implemented by a family of distributed protocols:

- **Stateless**: records, numbers, procedures, and classes. Since they do not change, these entities can be copied at will.[2] There is a trade-off between when to copy, how many times to copy to a site, and access time. This gives a family of protocols to define their distributed behaviors [7].

- **Single assignment**: logic variables. Assignment is done by a distributed unification algorithm, which is the subject of this paper. To be precise, logic variables provide *consistent multiple assignment*, that is, there can be multiple assignments as long as they are unifiable. We keep the phrase "single assignment" to avoid multiplying terminology.

- **Stateful**: cells, objects, ports, reentrant locks, and threads. For efficiency reasons, these entities' state pointers are localized to a site. If the state pointer's site can change, we say that the entity is mobile. Currently there are two mobility behaviors: a mobile state protocol (cells, objects, locks, ports) and a stationary access protocol (threads). The mobile state protocol ensures coherent state updates with controlled mobility of the state pointer [132]. The stationary access protocol is used for entities that cannot move.

- **Resource**: entities external to the shared store. References to resources can be passed around the network at will, but the resource can only be executed on its home site [130]. This includes computational and memory resources, which can be made visible in the language, e.g., by means of *virtual sites* [54].

The single-assignment category can be seen as an optimization of the stateful category in which a variable is bound to only one value, instead of repeatedly to different values. That is, the distributed unification algorithm is more efficient than the mobile state protocol. However, it turns out that logic variables have many more uses than simply as an optimization of stateful entities. These uses are explained below.

[2] This is true only for the entity; not for its external references. An external reference has its own protocol that corresponds to its category.

Logic variables

A logic variable conceptually has a fixed value from the moment of its creation. The value is unknown at first, and it remains unknown until the variable is bound. At all times, the variable can be used as if it were the value. If the value is needed, then the thread requiring the value will suspend until the variable is bound. If the value is not needed then execution continues.

A logic variable can be passed among sites arbitrarily. At all times, it "remembers its origins", that is, when the value becomes known then the variable will receive it. The communication needed to bind the variable is part of the variable, and not part of the program manipulating the variable. This means that the variable can be passed around at will, and the value will always arrive at the variable. This is one key reason why logic variables are useful in distributed computing.

Logic variables can replace standard (assignable) variables in all cases where they are assigned only one value, i.e., where they are used as placeholders for values. The algorithm used to bind logic variables must ensure that the result is independent of binding order. In a centralized system, the algorithm is called *unification* and is usually implemented as an extension of a union-find algorithm. Union-find handles only the binding of variables with variables [93]. Unification generalizes this to handle nonvariable terms as well. In a good implementation, binding a new variable to a nonvariable (the common case) compiles to a single register move or store operation [127].

A logic variable may be bound to another logic variable. A legitimate question is whether variable-variable binding is useful in a practical system. As we shall see, one reason that variable-variable binding is important is that it allows to maintain maximum latency tolerance and third-party independence when communicating among more than two sites, independent of fluctuating message delays. A second reason is that it has a very simple logical semantics.

It is possible to disallow variable-variable binding to obtain a slightly simpler implementation. The simpler implementation blocks any attempt to do variable-variable binding until at least one of the variables is bound to a value. The price of the simpler implementation is that third-party dependencies are not removed in all cases. Futures [49] and I-structures [12, 133, 64] resemble this weaker version of logic variables (see Section 8.10.2). There remains a crucial difference with logic variables, namely that futures and I-structures can be assigned only once, whereas logic variables can be assigned more than once, as long as the assignments are consistent with each other.

The efficiency difference between full and weak logic variables is small. The distributed binding algorithm is almost identical for the full and weak versions. Furthermore, the full version has a simple logical semantics. For these three reasons we have implemented the full version in the Distributed Oz implementation.

8.3.2 Distributed unification

For logic variables to be practical in a distributed setting they must be efficiently implementable. This section gives the key ideas of the distributed algorithm that realizes

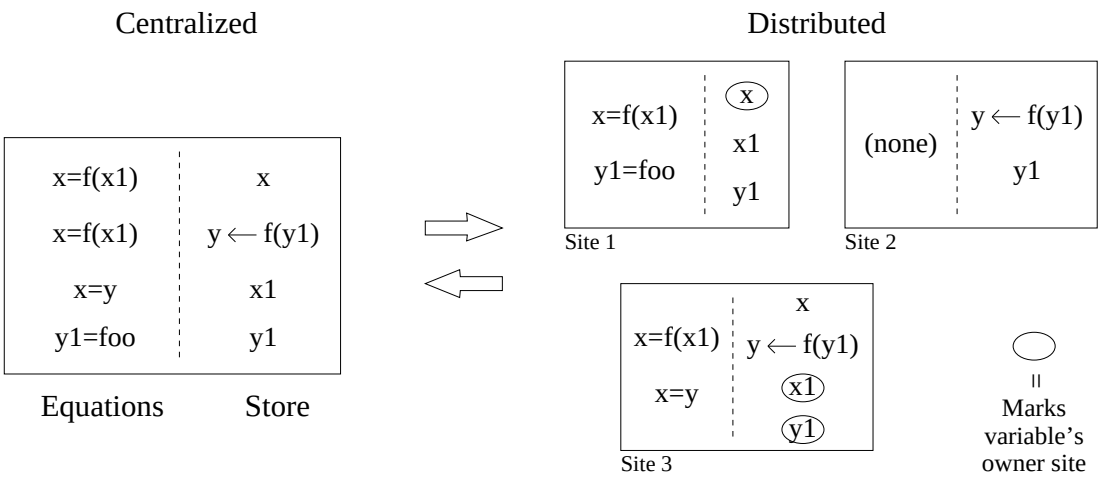


Figure 8.4: Initial configuration of example

this goal. We explain the algorithm in just enough detail so that its network behavior becomes clear. This will allow us to infer the algorithm's behavior in the programming examples that follow. A formal definition of the algorithm and proofs of its correct behavior are given starting from Section 8.4.

The two basic operations on logic variables are binding and waiting until bound. Waiting until bound is easy: the variable has a list containing suspended threads that need its value. When the value arrives, the threads are awoken. Binding is harder: it requires cooperation between sites. If a variable exists on several sites, then it must be bound to the same value on all sites, despite concurrent binding attempts. Unification implements the binding operation. At any instant there can be any number of bindings in various stages of completion. Both the centralized and distributed algorithms cause each binding request to be eventually incorporated into the store, if it is consistent with the store.

The basic distributed operation is binding a variable to a value. This is implemented by making one site the "owner" of the variable. In the current system, the site that declares the variable is its owner. A binding request is sent to the owner, and the owner forwards the binding to each site that knows the variable. In terms of network behavior, one message is sent to the owner, and one message is sent by the owner to each site that knows the variable. The owner's sends can be done by a reliable multicast, if the network supports it efficiently. The owner accepts the first binding request and ignores all subsequent binding requests. An ignored request will be retried by its initiating site after it receives the binding. As we will see in the programming examples, in the majority of cases a variable is declared either on a site that will need its value or on the site that will bind the variable. In both of these cases, the network behavior of the algorithm is very good.

A logic variable X can be bound to a data structure or to another variable. The algorithm is the same in both cases. By default the binding is *eager*, i.e., the new value

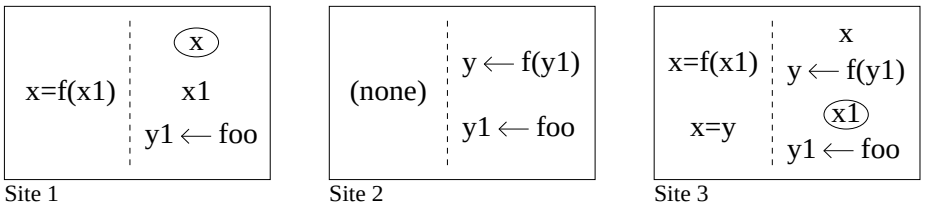


Figure 8.5: Configuration after executing the equation Y1=foo

is immediately sent to all sites that know about X. This means that a bound variable is guaranteed to eventually disappear from the system. The same binding eventually appears on each site that has the variable. For example, executing the equation X=f(X1) causes the binding X ← f(X1) to appear in the store on all sites containing X. Furthermore, X1 is added to the known variables of all of these sites that did not know X1.

An example

We illustrate the algorithm with an example. Figure 8.4 shows a centralized configuration (on the left) and one way to distribute it (on the right). Each configuration has a set of equations and a store. For the algorithm, an *equation* is simply a request to perform a binding. In the formal discussion (Section 8.4 and following), we need more kinds of requests than just equations. We call all the requests *actions*. The same equation may exist more than once. The store contains the variables and their bindings, if the latter exist.

In the distributed case, each site has a set of equations and a store. The centralized equations are distributed among the sites. Each variable is visible on a subset of the sites. If there is only one site, then the distributed algorithm is identical to the centralized algorithm. Each variable occurrence on a site is called a "proxy". One of the sites is the variable's owner. In Figure 8.4, site 1 is the owner of X and site 3 is the owner of both X1 and Y1. If the variable is bound, then the binding will eventually arrive on each site that sees the variable. Variable Y is bound to f(Y1) on sites 2 and 3.

Site 1 requests the binding Y1=foo. This sends a message to site 3, the owner of Y1. The owner sends a message to all proxies of Y1. That is, the owner sends three messages, to sites 1, 2, and 3. When a message arrives on a site, then the binding Y1 ← foo appears on that site (see Figure 8.5). Since the owner is on site 3, its message to site 3 does not need any network operations.

Lazy and eager variables

Logic variables can have different distributed behaviors, as long as network transparency is satisfied in each case. As explained above, by default a logic variable is *eager* on all sites, i.e., its binding is sent immediately to all sites that reference the vari-

able. This gives maximal latency tolerance and third-party independence. However, this may cause the binding to be sent to sites that do not need it. We say that a logic variable is *lazy* on a site if its value is only sent to that site when the site needs it, e.g., when a thread is waiting for the variable. Binding a lazy variable typically needs fewer messages, since not all sites that know the variable need its value. Both eager and lazy variables are implemented by the on-line DU algorithm of Section 8.6. They differ only in the scheduling of one reduction rule. The Mozart implementation currently only provides eager variables; with a minor change it can provide both. A programmer annotation can then decide whether a variable is eager or lazy. The implementation issues of laziness are further explored in Section 8.9.5.

8.3.3 Examples of concurrent programming

It turns out that logic variables suffice to express most concurrent programming idioms in an intuitive and concise manner. Additional concepts such as semaphores, critical sections, or monitors, are not needed. [15] conclude that logic variables are "spectacularly expressive" in concurrent programming even without explicit state. We give examples of four important idioms, namely synchronization, communication, mutual exclusion, and first-class channels. Many other idioms can be found in the concurrent logic programming literature [114, 51].

Synchronization and communication

The following fragment creates two threads and synchronizes their execution:

```
local X in
   thread {Print a} X=unit end
   thread {Wait X} {Print b} end
end
```

The statement `{Wait X}` blocks until `X`'s value is known. Therefore "`a`" is always printed before "`b`". The value of `X` is communicated from the first to the second thread. `{Wait X}` is not a new notion; it can be defined as **`if`** `X=1` **`then skip else skip end`**.

Mutual exclusion

A critical section can be defined by means of logic variables and one cell. The cell is used to manage access to a token, which is passed from one thread to the next. Assume that a cell `C` exists with initial content **`unit`**, e.g., defined by `{NewCell` **`unit`** `C}`. Then the following fragment defines a critical section:

```
local X Y in
   {Exchange C X Y} {Wait X} % Enter
   {InnerSanctum}            % Body
   Y=unit                    % Exit
end
```

We show that only one thread at a time can be executing the body. A thread that tries to enter is given C's previous state X_n and current state Y_n. The thread then waits on X_n. When the previous thread leaves, it binds Y_{n-1}=**unit**. Since $Y_{n-1} = X_n$, this allows the next thread to enter. This works even if many threads try to enter concurrently, since the exchanges are serialized. Section 8.3.4 uses this idea to define a procedure NewSimpleLock that can create any number of locks.

First-class channels

A simple kind of FIFO channel is the *stream*, a list with unbound tail. Reading from the stream is receiving from the channel. Appending to the stream is sending through the channel. Each element of the stream can be a record containing both the query and an answer channel. For example, here is a fragment that handles queries appearing on the stream X0:

```
case X0 of query(Q1 A1)|X1 then % Wait for query Q1 and channel
A1
   A1={CalcAnswerStream Q1}     % Calculate answer stream on A1
   case X1 of query(Q2 A2)|X2 then % Wait for Q1 and A2
      A2={CalcAnswerStream Q2}     % Calculate answer stream on
A2
      ...
   end
end
```

We assume that Q1 is a database query that gives multiple answers, which appear incrementally on A1. The **case** statement is a useful idiom: it waits until X0 is sufficiently bound to read the pattern query(Q1 A1)|X1. The pattern variables Q1, A1, X1 are declared implicitly. Typically, the above fragment would be written as part of a loop:

```
   local P in
      proc{P M}
         case M of query(Q A) then A={CalcAnswerStream Q} end
      end
      {ForAll X0 P}
   end
```

ForAll takes a list X0 and a one-argument procedure, and applies the procedure to all the list's elements. The above example can be written more compactly as a nested statement with exactly the same meaning:

```
   {ForAll X0
      proc{$ M}
         case M of query(Q A) then A={CalcAnswerStream Q} end
      end}
```

This is just a short-cut that avoids explicitly declaring P. Using ForAll is efficient; there are no memory leaks and the stream is not consumed faster than it is produced. It may be produced faster than it is consumed, however. Usually, a stream is associated to an Oz *port* and one writes to the port (see Section 8.3.1).

8.3.4 Examples of distributed programming

The purpose of this section is to show the usefulness of logic variables when extending concurrent object-oriented programming to a distributed setting. Sections 8.3.4-8.3.4 present a series of common programming idioms in distributed programming. We show how to express them in a concurrent object-oriented language that contains logic variables. The resulting solutions have two properties:

- The solutions perform correctly independently of how they are partitioned among sites. That is, the programming idioms underlying the communication patterns can be expressed in a network-transparent manner.
- No matter how the solutions are partitioned among sites, the resulting message traffic is optimal or near-optimal (in the common cases) or at least reasonable (in the uncommon cases). That is, given logic variables, the same programming idioms perform well in both centralized and distributed settings.

This shows that, at least in the cases presented here, using logic variables allows to keep useful programming idioms of centralized object-oriented programming, while allowing the implementation to extend efficiently to a distributed setting. This is evidence that controlling execution through data availability, which is what logic variables provide, is a natural way to keep good performance while mapping a program to arbitrary distribution structures.

The examples show a variety of distributed programming techniques using logic variables. Some of them, e.g., barrier synchronization and distributed locking, will normally be provided as primitives by the system. Others, e.g., stream communication, will normally be programmed by the user. We do not distinguish between the two cases since our goal is to show the expressiveness of logic variables.

Latency tolerance and third-party independence

From the viewpoint of execution order of basic language operations, a distributed execution cannot be distinguished from a concurrent execution. Distinguishing them requires looking at the effects of partitioning an execution over several sites. This affects system properties such as network properties (e.g., delays and limited bandwidth) and site resources (e.g., disks and main memory). At the language level, the latter shows up as the restriction of some operations to be local or remote only (such as local memory operations and remote message sends).

Logic variables decouple the declaration of a variable from its binding. Once a variable is declared, it can be passed to other sites, even before it is bound. When it is bound, the binding will be transferred automatically and efficiently to the sites needing it. This decoupling allows programs to provide a degree of *latency tolerance*, i.e., their execution is less affected by changes in network latency. For example, in the following fragment:

```
local Ans in
   thread
```

```
proc {Generate N Max L} % Return list of integers from N to
Max-1
   if N < Max then L1 in
      L=N|L1
      {Generate N+1 Max L1}
   else L=nil end
end

fun {Sum L A}           % Return (A + sum of elements of list
L)
   case L
   of nil then A
   [] X|Ls then {Sum Ls A+X}
   end
end

local CS L S in         % Generate a list and sum its elements
  CS={NewComputeServer ´sinuhe.sics.se´} % Remote compute
server
  thread L = {Generate 0 150000} end    % Producer thread
(local)
  {CS proc {$} S={Sum L 0} end}         % Consumer thread
(remote)
  {Print S}                             % Print result (local)
end
```

Figure 8.6: Stream communication

```
         {DataBase query("How far is up?" Ans)}
      end
      thread
         {RemoteClient inform(Ans)}
      end
   end
```

The database query and the client transfer are initiated concurrently. Assume that they are on different sites. The initiator site owns `Ans`. As soon as `Ans` is bound, the binding will be sent from the database site to the initiator site, which forwards it to the client site. This is all done independently of the initiator.

A logic variable can be bound to another logic variable. This allows programs to improve third-party independence. For example, assume variable `X` exists on sites 1 and 2 and variable `Y` exists on sites 2 and 3. Assume that `X` and `Y` are bound together on site 2. Then binding `X` to 23 on site 1 should be visible on site 3 independent of what happens to site 2.

```
proc {Generate N L} % Return list L of integers starting with
N
   case L of X|Ls then
      X=N
      {Generate N+1 Ls}
   else skip end
end

fun {Sum N L A}     % Return (A + sum of first N elements of
L)
   if N>0 then X L1 in
      L=X|L1
      {Sum N-1 L1 A+X}
   else
      A
   end
end

local CS L S in
  CS={NewComputeServer ´sinuhe.sics.se´} % Remote compute
server
  {CS proc {$} {Generate 0 L} end}      % Producer thread
(remote)
  thread S={Sum 150000 L 0} end         % Consumer thread
(local)
  {Print S}                             % Print result (local)
end
```

Figure 8.7: Stream communication with flow control

Stream communication

The first example program consists of a producer that creates a data stream and a consumer that reads this stream. We first examine the program in a centralized setting. Then we explain what happens when the producer and consumer are on different sites.

As we saw before, a stream is a list whose tail is a logic variable. The producer thread incrementally binds the tail to a pair of an element and a new tail. The consumer thread can start reading the stream while the producer is still writing to it. In the program of Figure 8.6, the producer generates a list of numbers and the consumer sums them. The consumer will print the sum 11250075000. This example has no flow control, i.e., the producer will create elements eagerly. Unless the list's maximum size is small, flow control is needed to avoid problems with memory utilisation. This is true in both centralized and distributed settings.

Flow control can be added by letting the consumer inform the producer when it is able to receive the next element. We do this by letting the consumer bind the tail to a pair of a logic variable and a new tail. The producer waits until this pair exists and then binds the logic variable to the next element. The producer and consumer then execute in lock step (see Figure 8.7). An *n*-element buffer can be programmed with minor changes. The consumer can terminate the producer if the **`skip`** is replaced with `L=nil`. The producer detects the end of `L` and terminates.

To distribute these examples, let the producer thread and consumer thread run on different sites. For example, Figure 8.6 executes the producer locally and the consumer remotely, and Figure 8.7 does it the other way around. In both cases, remote execution is initiated by designating the site on which a calculation starts.

The distributed behavior is as follows. In the first example, the consumer is started by sending a request, i.e., a zero-argument procedure, to a remote compute server `CS`. This procedure and the function `Sum` are defined locally. Their compiled code is sent across the network. Because of network transparency, the procedure body can be any expression at all. The logic variable `S` is shared between the local and remote sites, and therefore transparently becomes a distributed variable. When `Sum` finishes its calculation then the result is bound to `S`. This sends the result back to the local site.

In terms of network operations, both examples are efficient. Since `N` is already bound when `L` is bound, Figure 8.6 will send one message from the producer to the consumer for each element of the stream, exactly as desired. Figure 8.7 will send one message in each direction for each element of the stream, exactly as desired.

Stream communication with multiple readers

Now let the stream be read by multiple consumers. Figure 8.8 shows how to do it with consumers on three sites. We assume three compute servers referenced by `CS1`, `CS2`, and `CS3`. Both examples of the previous section (with and without flow control) will work with multiple consumers. This is an excellent illustration of the difference between logic variables and I-structures. It is allowed for multiple readers to bind the list's tail, since they bind it in a consistent way. This would not work with ordinary single assignment, e.g., as provided by I-structures.

```
local CS1 CS2 CS3 L S1 S2 S3 in
   CS1={NewComputeServer ´sinuhe.sics.se´}
   CS2={NewComputeServer ´norge.info.ucl.ac.be´}
   CS3={NewComputeServer ´tinman.ps.uni-sb.de´}
   thread {Generate 0 L} end              % Producer (local)
   {CS1 proc {$} S1={Sum L 150000 0} end} % Consumer 1 (on
Site 1)
   {CS2 proc {$} S2={Sum L 150000 0} end} % Consumer 2 (on
Site 2)
   {CS3 proc {$} S3={Sum L 150000 0} end} % Consumer 3 (on
Site 3)
end
```

Figure 8.8: Stream communication with multiple readers

The example without flow control is straightforward: one message is sent to each consumer per element. The example with flow control is more interesting; it is shown in Figure 8.8. In this case, each consumer sends a message to request the next element when it is needed. The network behavior is as follows. To make things interesting, we assume a fast, a medium, and a slow consumer. The fast consumer sends a message to the producer, which is the owner of the first stream variable. The message contains two variables: one for the element and one for the next stream variable. Both of these variables are owned by the fast consumer. It follows that from this point on, the fast consumer will be the owner of all stream variables. Therefore all further stream elements will be sent by the producer to the fast consumer, who will multicast them to the other consumers. After the first message, the medium consumer will send requests to the fast consumer, since it is the owner. These requests will be ignored, since the fast consumer will already have bound the stream variable. The slow consumer will send no requests at all; it receives the elements before asking for them.

Barrier synchronization

We would like to create a set of concurrent tasks and be informed as soon as all tasks have finished. This should work efficiently independently of how the tasks are partitioned over a set of sites. Figure 8.9 gives a simple solution that works well in both centralized and distributed settings. To explain how it works, we need first of all to understand how to synchronize on the termination of a single thread. This is done as follows, where statement `S` represents a task:

```
local X in thread S X=unit end ... {Wait X} end
```

The main thread creates a new thread whose body is `S X=unit`. The new thread will bind `X` after `S` is finished, and the main thread detects this with a `{Wait X}`. A statement `S` *finishes* when it reduces to **`skip`** in its thread. Other threads may be created

```
proc {BarrierSync Ps}
   proc {Conc Ps L}
      case Ps of P|Pr then X Ls in
         L=X|Ls
         thread {P} X=unit end
         {Conc Pr Ls}
      else
         L=nil
      end
   end
   L
in
   {Conc Ps L}
   {ForAll L proc {$ X} {Wait X} end}
end

{BarrierSync [proc {$} E1 end   % Task 1
              proc {$} E2 end   % Task 2
              proc {$} E3 end]} % Task 3
```

Figure 8.9: Barrier synchronization

during the execution of S; these are independent of S. If the task is executed remotely, then binding X sends a single message to the main site, which owns X. This informs the thread of the task's completion. The message sent back to the task's site is a simple acknowledgement that does not affect the barrier's latency, which is one message.

We generalize this idea to multiple tasks. The general scheme is as follows:

```
local X1 ... Xn in
   thread S1 X1=unit end
   thread S2 X2=unit end
   ...
   thread Sn Xn=unit end
   {Wait X1} ... {Wait Xn} S
end
```

The main thread waits until all Xi are bound. When Si terminates then its thread binds Xi=**unit**. When all tasks terminate then all Xi are bound, so the main thread runs S.

Assume now that the tasks are distributed over a set of sites. Each Xi is owned by the main thread's site. Therefore binding Xi=**unit** sends a message from the task site to the main site. When all variables are bound, the main thread resumes execution. Concurrently, the main site sends a message back for each message it received. These messages do not affect the barrier's latency.

```
proc {NewSimpleLock ?Lock}
   Cell = {NewCell unit}
in
   proc {Lock Code}
      Old New in
      try
         {Exchange Cell Old New} {Wait Old}  % Enter
         {Code}                              % Body
      finally New=unit end                   % Exit
   end
end
```

Figure 8.10: Distributed locking

Distributed locking

If a program fragment may be executed by many threads, then it is important to be able to guarantee mutual exclusion. A thread that attempts to execute the fragment should block and be queued. Multiple requests should be correctly queued and blocked independent of whether the threads are on the same site or on another site. We show that it is possible to implement this concisely and efficiently in the language. As explained in Section 8.3.3, Figure 8.10 shows one way to implement a lock that handles exceptions correctly.[3] If multiple threads attempt to access the lock body, then only one is given access and the others are queued. The queue is a sequence of logic variables. Each thread blocks on one variable in the sequence, and will bind the next variable after it has executed the lock body. Each thread desiring the lock therefore references two logic variables: one to wait for the lock, and one to pass the lock to the next thread. Each logic variable is referenced by two threads.

If the threads are on different sites, then the queue is distributed. A single message will be sent to transfer the lock from one site to another. This implements distributed token passing, which is a well-known distributed algorithm for mutual exclusion [25]. We explain how it works. When a thread tries to enter the lock body, the `Exchange` gives it access to the previous thread's `New` variable. The previous thread's site is `New`'s owner. When the previous thread binds `New`, the owner sends the binding to the next thread's site. This requires a single message.

Remote method invocation (RMI)

Let us invoke an object from within a thread on a given site. Where will the object execute? On a network-transparent system there are several possible answers to this question. Here we give just enough information to justify our RMI implementation. For a full discussion of the issues we refer the reader to [131, 132]. In Mozart, objects

[3] A thread-reentrant lock is defined in [132].

```
proc {NewStationary Class Init ?StatObj}
   Obj={New Class Init}
   S P={NewPort S}
   N={NewName}
in
   proc {StatObj M}
      R in
      {Send P M#R}
      if R=N then skip
      else raise R end
      end
   end
   thread
     {ForAll S
      proc {$ M#R}
         thread
            try {Obj M} R=N
            catch X then R=X end
         end
      end}
   end
end
```

Figure 8.11: RMI definition: Create a stationary object from any class

```
% Create class Counter on local site
class Counter
   attr i
   meth init i <- 0 end
   meth inc  i <- @i+1 end
   meth get(X) X=@i end
   meth error raise e(some_error) end end
end

% Create object Obj on remote site
{CS proc {$} Obj={NewStationary Counter init} end}

% Invoke object from local site
{Obj inc}
{Obj inc}
local X in {Obj get(X)} {Print X} end
try {Obj error} catch X then {Print X} end
```

Figure 8.12: RMI example: A stationary counter object

synchronously migrate to the invoking site by default. Therefore the object executes locally with respect to the invoking thread. This makes it easy for the object to synchronize with respect to the thread. If the object raises an exception, then it is passed to the thread. Object migration is implemented by a lightweight mobility protocol that serializes the path of the object's concurrent state pointer among the invoking sites.

It is possible in Oz to define a generic procedure that takes any object and returns a stationary object, i.e., such that all its methods will execute on the same site. This works because Oz has first-class messages and dynamic typing [58]. This is not possible in Java [45]. Figure 8.11 defines `NewStationary`, which given any object class, creates a stationary object of that class. It works by wrapping the object inside a port, which is a stationary entity to which messages can be sent asynchronously. Therefore the object always executes on the same site, namely the site on which it was created. As before, the object synchronizes with respect to the invoking thread, and exceptions are passed to the invoking thread. The logic variable `R` is used both to synchronize and to pass back exceptions.

Figure 8.12 defines `Obj` remotely and invokes it from the local site. For example, `{Obj get(X)} {Print X}` queries the object and prints the result on the local site. The object responds by binding the variable `X` with the answer. Since the local site owns `X`, the binding request sends one message from the remote site to the local site. With the initial invocation, this gives a total message latency of two for the remote call, just like an RPC. There is a third message back to the remote site that does not affect the message latency.

```
public class List {
   final unknown int car;
   final unknown List cdr;

   List(unknown int car, unknown List cdr) {
      this.car=:=car;
      this.cdr=:=cdr;
   }
   public void cons(unknown int car, unknown List cdr) {
      this.car=:=car;
      this.cdr=:=cdr;
   }
}
```

Figure 8.13: List implementation in CC-Java

8.3.5 Adding logic variables to other languages

This section shows how to add logic variables in an orthogonal way to Java and ML, representative examples of object-oriented and functional languages.

Java

[121] has recently defined and implemented a Java variant, CC-Java (Concurrent Constraint Java), which replaces monitors by logic variables and adds statement-level thread creation. Except for these differences, CC-Java has the same syntax and semantics as Java.

CC-Java provides logic variables through a single new modifier, `unknown`, which can be used in declarations of local variables, fields, formal parameters, and functions. For example, a variable `i` declared as `unknown int i;` is initially assigned an unknown value. Standard Java variables can be freely replaced by unknown variables. The result is always a legal CC-Java program. Variables with Java types will never be assigned unknown values–any attempt will suspend the thread until the value is known.

An `unknown` variable is bound by the new operator "`=:=`", which does unification. Each of the two operands can be known (i.e., be a standard Java variable) or unknown. Doing `i=:=23` binds `i` to `23`. For correctness, the assignment operator "`=`" must overwrite (not unify) an `unknown` variable on the left-hand side. Declaring an `unknown` variable as `final` means that it is only assigned once, i.e., when it is declared. An `final unknown` variable is therefore equivalent to an Oz logic variable. An `unknown` variable is equivalent to an Oz cell that points to a logic variable.

Figure 8.13 shows how to implement lists in CC-Java. Each list pair contains two logic variables, and therefore lists can be partially instantiated just like in Oz. Using

```
public class StreamExample {
   // Return list of integers from n to max-1
   static List generate(int n, int max) {
      final unknown List l;
      unknown List ptr=l;
      for (int i=n; i<max; i+=1) {
         final unknown List tail;
         ptr=:=new List(i,tail);
         ptr=tail;
      }
      ptr=:=null;
      return l;
   }

   // Return (a + sum of elements of list l)
   static int sum(unknown List l, int a) {
      int sum=a;
      unknown List ptr=l;
      while (ptr!=null) {
         final unknown int x;
         final unknown List ls;
         ptr.cons(x,ls);
         sum+=x;
         ptr=ls;
      }
      return sum;
   }

   // Generate a list and sum its elements
   public static void main(String[] args) {
      unknown List l;
      int sum;
      thread l=:=generate(0,1500);
      sum=sum(l,0);
      System.out.println(sum);
   }
}
```

Figure 8.14: Stream communication in CC-Java

logic variables does not imply any memory penalty for lists: when compiled to Oz, a CC-Java list pair uses just two memory words. Threads can synchronize on the instantiation state of lists.

Figure 8.14 uses these lists to write the stream communication example of Figure 8.6 in CC-Java (see Section 8.3.4). The `thread` statement of CC-Java is used to generate the list in another thread. The example has been written in a natural style in Oz and CC-Java, where Oz uses recursion and CC-Java uses iteration to define the `generate` and `sum` functions. Comparing the two examples, we see that there is very little difference in clarity between these two styles. Their run-time efficiencies are comparable.

When examining the CC-Java program, two observations can be made. First, the example has two synchronization points: the statements `ptr.cons(x,ls)` and `sum+=x` inside the `sum` function. The former waits until `ptr` contains a list pair and the latter waits until `x` is an integer. Second, the example shows that both `final unknown` and `unknown` variables are useful. The former are used as part of data structures that should not change. The latter are used in loops that need a new logic variable in each iteration. The new variable is created with an assignment statement, e.g., `ptr=ls` creates a new variable referred to by `ptr`.

It is straightforward to compile CC-Java to either Oz or Java. A prototype CC-Java to Oz compiler has been implemented that understands the full Java syntax and compiles most of the Java language. Benchmarks show that CC-Java and Oz have comparable performance on the Mozart implementation of Distributed Oz. Both CC-Java and Oz on Mozart have performance comparable to Java on JDK 1.1.4, except that threads are much faster in Mozart [59, 58].

We outline how to implement a CC-Java to Java compiler. All Java code that does not use logic variables is unchanged. For each class `C` of which `unknown` instances are declared, the compiler adds a second class definition `UnknownC` to the Java code. The class `UnknownC` includes all methods of `C` and additional methods to unify the variable and to obtain its value. At each point where the value of an object of class `UnknownC` is needed, the compiler inserts a call to obtain the value. If the value is not yet available, then the calling thread is suspended until the value becomes available through unification.

ML

[117] has recently shown how logic variables can be added as a conservative extension to a functional language very similar to Standard ML. We outline how the extension is done. Several new operations are added, including the following:

- `lvar: unit -> 'a`. The operation `lvar()` creates a fresh logic variable and returns its address.
- `<-: 'a * 'a -> 'a`. The operation `x <- y` binds `x`, which must be a logic variable, to `y`.

- `==: 'a * 'a -> 'a`. The operation `x == y` unifies `x` and `y`. This raises an exception if `x` and `y` are not unifiable.
- `wait: 'a -> 'a`. The operation `wait x` is an identity function that blocks until its argument is nonvariable.
- `spawn e`. This operation spawns a new thread evaluating expression *e* and returns `()`.

Execution states map addresses to any ML entity including primitive values, records, and reference cells. Execution states are extended so that a state may also map an address to a logic variable or to another address. The entity corresponding to an address is obtained by iterating the state function until the result is no longer an address. This iteration is the dereferencing operation. If a thread needs an entity and encounters a logic variable, then it blocks until the entity is available.

With this extension, existing ML programs continue to work and logic variables may be freely intermixed with ML entities. ML provides explicit stateful entities which are called *references* and behave like typed Oz cells. As in Oz and CC-Java, the combination of logic variables and state allows to easily express powerful concurrent programming techniques. Smolka outlines the semantics of the extension and illustrates some of these programming techniques.

8.4 Basic concepts and notation

This section introduces the basic concepts and notation used for the CU and RCU algorithms, which do centralized unification. Most of this notation remains valid for the distributed algorithms. The extra notation they need will be given later on.

8.4.1 Terms and constraints

In the usual case, a variable will be bound to a data structure. However, because of unpredictable network behavior, it may also be necessary to bind variables to variables or data structures to data structures. The result should not depend on the order in which the bindings occur. This justifies using a constraint system (*D*,*C*) to model the data structures and their bindings [67]. The domain *D* is the set of data structures of interest; for generality we assume these are rational trees, i.e., rooted directed graphs. The constraints *C* model bindings; we assume they are equalities between terms that describe sets of trees. For example, the constraint $x = f(y)$ means that the trees described by the variable *x* all have a root labelled f and a single subtree, which is a tree described by the variable *y*. In this way, we express clearly what it means to bind terms that may contain unbound variables. If *y* is not bound, then nothing is known about the subtree of *x*.

We introduce a uniform notation for terms, which can be either variables or trees that may contain variables. Terms are denoted by *u*, *v*, *w*. Variables are denoted by *x*,

y, z. Nonvariable terms are denoted by t, t_1, t_2. A term can either be a variable or a nonvariable. A nonvariable term is a record of the form $f(x_1,...,x_n)$ with arity $n \geq 0$, where x_1, ..., x_n are variables, and where the label f is an atomic constant taken from a given set of constants. A constraint has the general form $\bigwedge_i u_i = v_i$ where u_i and v_i are terms. A *basic* constraint has the form $x = u$.

To bind u and v means to add the constraint $u = v$ to the system. This is sometimes called *telling* the constraint. The operation of binding u and v is called *unification*. This is implementable in a time and space essentially equivalent to that needed for manipulating data structures in imperative languages [127]. For more on the constraint-solving aspects of unification see [67].

For the purpose of variable-variable unification, we assume a partial order between terms such that all variables are in a total order and all nonvariable terms are less than all variables. That is, we assume a transitive antisymmetric relation less(u,v) such that for any distinct variables x and y, exactly one of less(x,y) or less(y,x) holds. In addition, for any nonvariable term t and any variable x, less(t,x) holds. The algorithm uses the order to avoid creating binding cycles (e.g., x bound to y and y bound to x). This is especially important in a distributed setting.

8.4.2 Configurations

A *configuration* $c = (\alpha;\sigma;\mu)$ of a centralized execution is a triple containing an action α, a store σ, and a memo table μ:

$$\alpha = \bigwedge_i u_i = v_i \wedge \bigwedge_i \text{false} \wedge \bigwedge_i \text{true} \qquad \sigma = \bigcup_i x_i \leftarrow u_i \qquad \mu = \bigcup_i x_i = y_i$$

The action α is a multiset of three kinds of primitive actions, of the form $u = v$, false, and true. The equation $u = v$ is one kind of primitive action. The notation $x \leftarrow u$ represents the binding of x to u. The store is a set of bindings. All variables x_i in the store are distinct and there are no cycles $x_{a_1} \leftarrow x_{a_2}$, ..., $x_{a_{n-1}} \leftarrow x_{a_n}$, $x_{a_n} \leftarrow x_{a_1}$. It is easy to show that configurations always have this form in the CU algorithm.

The notation $x \leftarrow u, \sigma$ will be used as shorthand for $\{x \leftarrow u\} \cup \sigma$. The function $\text{lhs}(\sigma) = \bigcup_i x_i$ gives the set of bound variables in σ, which are exactly the variables on the left-hand sides of the binding arrows.

The memo table μ is used to store previously-encountered variable-variable equalities so that the algorithm does not go into an infinite loop when unifying terms representing cyclic rational trees. For example, consider the equation $x = y$ with store $x \leftarrow f(x) \wedge y \leftarrow f(y)$. Dereferencing x and y and decomposing the resulting equation $f(x) = f(y)$ gives $x = y$ again (see Section 8.5). This loop is broken by putting $x = y$ in the memo table and testing for its presence. The memo table has two operations, *ADD* and *MEM*, defined as follows:

$$\begin{array}{lcl} ADD(x = y, \mu) & = & \mu \cup \{x = y\} \\ ADD(x = t, \mu) & = & \mu \\ MEM(x = u, \mu) & = & \text{true if } x = u \in \mu \\ MEM(x = u, \mu) & = & \text{false otherwise} \end{array}$$

If the number of variables is finite, then the number of possible variable-variable equations in the memo table is finite also. Therefore all possible loops are broken. In the off-line case this is always true. In the on-line case this is true if the finite size property holds (see Section 8.8).

8.4.3 Algorithms

We define an algorithm as a set of reduction rules, where a rule defines a transition relation between configurations. The algorithms in this article all have a straightforward translation into an efficient imperative pseudocode. We do not define the algorithms in this way since it complicates reasoning about them. Rule reduction is an atomic operation. If more than one rule is applicable in a given configuration, then one is chosen nondeterministically. A rule is defined according to the following diagram:

$$\frac{\alpha \;\|\; \alpha'}{\sigma;\, \mu \;\|\; \sigma';\, \mu'} \quad C$$

A rule becomes applicable for a given action α when the actual store matches the store σ given in the rule and the optional condition C is satisfied. The rule's reduction atomically replaces the current configuration $(\alpha;\sigma;\mu)$ by the result configuration $(\alpha';\sigma';\mu')$.

Both the centralized and the distributed algorithms are defined in the context of a structure rule and a congruence.

$$\text{Structure} \quad \frac{\alpha_1 \wedge \alpha_2 \;\|\; \alpha_1' \wedge \alpha_2}{\sigma;\, \mu \;\|\; \sigma';\, \mu'} \quad \textbf{if} \quad \frac{\alpha_1 \;\|\; \alpha_1'}{\sigma;\, \mu \;\|\; \sigma';\, \mu'}$$

$$\text{Congruence} \quad \begin{cases} \alpha_1 \wedge \alpha_2 \equiv \alpha_2 \wedge \alpha_1 \\ \text{true} \wedge \alpha \equiv \alpha \end{cases}$$

Because of the congruence, the primitive action true may occur an arbitrary number of times. This is not true of the other primitive actions, which form a multiset.

8.4.4 Executions

An *execution e* of a given algorithm is a (possibly infinite) sequence of configurations such that the first configuration is an initial configuration and each transition corresponds to the reduction of one rule:

$$c_1 \xrightarrow{R_1} c_2 \xrightarrow{R_2} \cdots \xrightarrow{R_{n-1}} c_n$$

In any execution, distributed or centralized, we assume that the rules are reduced in some total order. This is done without loss of generality since the results we prove in this article will hold for *all* possible executions. Therefore, reductions that are not causally related may be assumed to execute in any order.

An *initial* configuration of the CU algorithm is of the form $(\alpha_1;\emptyset;\emptyset)$, where α_1 is a finite conjunction of equations and the store and memo table are both empty. A *terminal* configuration (if it exists) is a configuration where no rules are applicable. The

last configuration of a finite execution is not necessarily a terminal configuration; it is simply the configuration that has no successors. A *valid* configuration is one that is reachable by an execution of a given algorithm. We speak of centralized executions (using the CU or RCU algorithms, see Sections 8.5 and 8.7.2) and distributed executions (using the DU algorithm, see Section 8.6).

8.4.5 Adapting unification to reactive systems

A store represents a logical conjunction of constraints. Adding a constraint that is inconsistent with the store results in the conjunction collapsing to a "false" state. This behavior is incompatible with a long-lived reactive system. Furthermore, it is expensive in a distributed system since it requires a global synchronization. Rather, we want an inconsistent constraint to be flagged as such (e.g., by raising an exception), without actually being put in the store. This requires two modifications to the unification algorithm:

- **Incremental tell**, i.e., information that is inconsistent with the store is not put in the store [116]. The CU and DU algorithms both implement incremental tell by decomposing a rational tree constraint into the basic constraints $x = y$ and $x = t$ and by not collapsing the store when an inconsistency is detected with a basic constraint. Inconsistency is represented as a new action "false" instead of being incorporated into the store. This new action can be used to inform the program, e.g., by raising an exception.

- **Full failure reporting**, i.e., every case of inconsistency is flagged to the program. Neither the CU nor the DU algorithms do this. If the inconsistent equation $x = y$ is present more than once, then the CU algorithm flags this only once. The DU algorithm flags the inconsistency of $(x = y)_s$ only once per site s. That is, both algorithms are guaranteed to flag an inconsistency only once per memo table. Inconsistencies can be flagged more often by introducing more memo tables. This may sometimes do redundant work. For example, if equations reduce within a thread and each thread is given its own memo table, then multiple occurrences of an inconsistent equation will be flagged once per thread. The Mozart implementation will flag an inconsistency at least once per program-invoked unification (see Section 8.9).

8.5 Centralized unification (CU algorithm)

This section defines the CU algorithm, an algorithm for rational tree unification. The definition is given as a set of transition rules with an interleaving semantics. That is, rule reductions do not overlap in time. Only their order is important. Nothing is lost by using an interleaving semantics, since we prove properties that are true for all interleavings that are valid executions [5].

The CU algorithm is the base case for the DU algorithm of Section 8.6. There are many possible variant definitions of unification. The definition presented here is deterministic and constrained by the requirement that it must be extensible to a practical distributed algorithm. That is, the use of global conditions is minimized (see Section 8.6.1).

8.5.1 Definition

We define the CU algorithm by the following seven rules.

$$\textsc{Interchange}\ \frac{u = x \;\|\; x = u}{\sigma;\ \mu \;\|\; \sigma;\ \mu} \quad \text{less}(u,x)$$

$$\textsc{Bind}\ \frac{x = u \;\|\; \text{true}}{\sigma;\ \mu \;\|\; x \leftarrow u, \sigma;\ \mu} \quad \text{less}(u,x), x \notin \text{lhs}(\sigma)$$

$$\textsc{Memo}\ \frac{x = u \;\|\; \text{true}}{x \leftarrow v, \sigma;\ \mu \;\|\; x \leftarrow v, \sigma;\ \mu} \quad \text{less}(u,x),\ MEM(x = u, \mu)$$

$$\textsc{Dereference}\ \frac{x = u \;\|\; v = u}{x \leftarrow v, \sigma;\ \mu \;\|\; x \leftarrow v, \sigma;\ ADD(x = u, \mu)} \quad \text{less}(u,x),\ \neg MEM(x = u, \mu)$$

$$\textsc{Identify}\ \frac{x = x \;\|\; \text{true}}{\sigma;\ \mu \;\|\; \sigma;\ \mu}$$

$$\textsc{Conflict}\ \frac{t_1 = t_2 \;\|\; \text{false}}{\sigma;\ \mu \;\|\; \sigma;\ \mu} \quad t_1 = f_1(x_1,...,x_m),\ t_2 = f_2(y_1,...,y_n),\ (f_1 \neq f_2 \vee m \neq n)$$

$$\textsc{Decompose}\ \frac{t_1 = t_2 \;\|\; \bigwedge_{1 \leq i \leq n} x_i = y_i}{\sigma;\ \mu \;\|\; \sigma;\ \mu} \quad t_1 = f(x_1,...,x_n),\ t_2 = f(y_1,...,y_n)$$

8.5.2 Properties

To prove total correctness of the on-line algorithm, we need the following entailment property of the CU algorithm. We will prove that the DU algorithm implements the CU algorithm. It follows that the entailment property also holds for the DU algorithm.

Logical formula of a configuration. A configuration $c = (\alpha;\sigma;\_)$ has an associated logical formula $\varepsilon(c) = \varepsilon_a(\alpha) \wedge \varepsilon_s(\sigma)$, where:

$$\begin{array}{lcl}
\varepsilon_a(\alpha_1 \wedge \alpha_2) & = & \varepsilon_a(\alpha_1) \wedge \varepsilon_a(\alpha_2) \\
\varepsilon_a(u = v) & = & u = v \\
\varepsilon_a(\text{true}) & = & \text{true} \\
\varepsilon_a(\text{false}) & = & \text{false} \\
 & & \\
\varepsilon_s(\sigma_1 \cup \sigma_2) & = & \varepsilon_s(\sigma_1) \wedge \varepsilon_s(\sigma_2) \\
\varepsilon_s(\{x \leftarrow u\}) & = & x = u
\end{array}$$

Theorem 8.5.1 ((Logical equivalence property)). *In every transition $c_i \rightarrow c_{i+1}$ of every execution of the CU algorithm, the logical equivalence $\varepsilon(c_i) \leftrightarrow \varepsilon(c_{i+1})$ holds under the standard equality theory.*

Proof. By standard equality theory we mean the theory $\mathcal{E}$ given by [84] (page 79), minus the acyclicity axioms (i.e., rule 4). This theory has the usual axioms implying nonequality of distinct symbols, term equality implies argument equality and vice versa, substitution by equals is allowed, and identity. What we want to prove is $\mathcal{E} \models \forall \varepsilon(c_i) \leftrightarrow \varepsilon(c_{i+1})$, where the quantification is over all free variables. This is a standard result in unification theory [50, 26, 87, 52]. □

cor 1 ((Entailment property or CU total correctness)). *Given any initial configuration $c_1 = (\alpha_1;\emptyset;\emptyset)$ of the CU algorithm. Then the algorithm always reaches a terminal configuration $c_n = (\alpha_n;\sigma_n;\_)$ with $\varepsilon(c_n) \leftrightarrow \varepsilon(c_1)$. Furthermore, α_n consists of zero or more false actions, and if there are zero, then $\varepsilon_s(\sigma_n) \leftrightarrow \varepsilon_a(\alpha_1)$.*

Proof. Again, this is a standard result in unification theory. The equivalence follows from the previous theorem. □

8.6 Distributed unification (DU algorithm)

This section defines a distributed algorithm for rational tree unification. The section is organized as follows. Section 8.6.1 explains how to generalize the CU algorithm to a distributed setting. Section 8.6.2 sets the stage by giving the basic concepts and notation needed in the DU algorithm. Section 8.6.4 defines the DU algorithm in two parts: the non-bind rules, which are the local part of the algorithm, and the bind rules, which are the distributed part. Finally, Section 8.6.5 compares the CU and DU algorithms from the viewpoint of the dereference operation.

8.6.1 Generalizing CU to a distributed setting

A distributed algorithm must be defined by reduction rules that do local operations only, since these are the only rules we can implement directly. To be precise, two conditions must be satisfied. First, testing whether a rule is applicable should require looking only at one site. Second, reducing the rule should modify only that site, except that the rule is allowed to create actions annotated with other sites. In the distributed system these actions correspond to messages. Rules that satisfy these two conditions are called *local* rules. A distributed algorithm defined in terms of local rules is a *transition system* in an asynchronous non-FIFO network [124].

We would like to extend each CU rule to become a local rule in the distributed setting. In this way, we maintain a close correspondence between the centralized and distributed algorithms, which simplifies analysis of the distributed case. Furthermore, this minimizes costly communication between sites.

The first step is to annotate the rule's actions and bindings with sites. Each CU rule reduces an input action and may inspect a binding in the store. We annotate the input

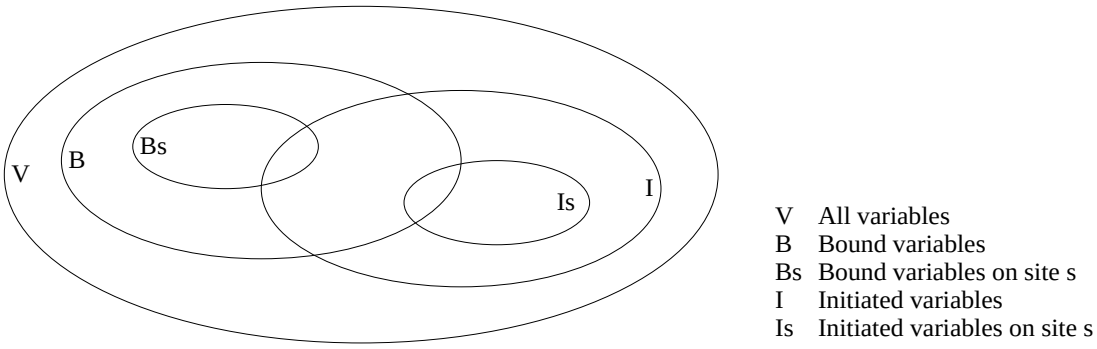


Figure 8.15: Bound variables and initiated variables

action by its site and the binding by the same site. This is correct if we assume that a binding will eventually appear on each site that references the variable. We annotate the output action by the same site as the input action. A "true" output action does not need a site. Actions may remain unannotated, in which case the DU algorithm does not specify where they are reduced. This set of annotations suffices for the rules INTERCHANGE, IDENTIFY, CONFLICT, and DECOMPOSE to become DU rules. An important property of CONFLICT is that an inconsistency is always flagged on the site that causes it.

The three remaining CU rules cannot be so easily extended since they have global conditions. To be precise, BIND has the unboundness condition $x \notin \mathrm{lhs}(\sigma)$,[4] and MEMO and DEREFERENCE both have the memoization condition $MEM(x = u, \mu)$. It turns out that the memoization condition can be relaxed in the distributed algorithm, so that it becomes a local condition there. In this way, the MEMO and DEREFERENCE rules become local rules. The idea is to give each site its own memo table, which is independent of the other memo tables. Section 8.7.2 proves that this relaxation is correct, but that redundant local work may be done.

The unboundness condition of BIND cannot be eliminated in this way. Implementing it requires communication between sites. The single BIND rule therefore becomes *several* local rules in the distributed setting. The BIND rule is replaced by four rules that exchange messages to implement a coherent variable elimination algorithm.

The resulting DU algorithm consists of ten local rules, namely six non-bind rules (Section 8.6.4) and four bind rules (Section 8.6.4). The six non-bind rules do not send any messages. Of the four bind rules, only INITIATE and WIN send messages. All rules test applicability by looking at one site only, except for WIN and LOSE, which use information tied to a variable but not tied to any particular site, namely a flag $\mathrm{unbound}(x)/\mathrm{bound}(x)$ and a binding request $(x \sim u)$.

[4] The opposite condition, confirming the existence of a binding, is local.

8.6.2 Basic concepts and notation

We introduce a set $S = \{1, ..., k\}$ of k sites, where $k \geq 1$. We model distributed execution by *placing* each action and each binding on a site. A primitive action or binding ξ is placed on site s by adding parentheses and a subscript $(\xi)_s$. The same ξ may be placed on several sites, in which case the resulting actions or bindings are considered separately. A configuration of a distributed execution is a triple $(A;\Sigma;M)$ consisting of an action A, a store Σ, and a memo table M. We denote the action, store, and memo table on site s by A_s, Σ_s, and M_s, respectively.

Store

The store Σ contains sited bindings $(x \leftarrow u)_s$, sited binding initiations $(x \leftarrow \bot)_s$, and flags to denote whether a variable is bound or not (bound(x) or unbound(x)). A store has the form:

$$\begin{aligned}
\Sigma &= \Gamma \cup \bigcup_{s \in S} \Sigma_s \\
\Sigma_s &= \bigcup_{x_i \in B_s} (x_i \leftarrow u_i)_s \cup \bigcup_{x_i \in I_s} (x_i \leftarrow \bot)_s \\
\Gamma &= \bigcup_{x_i \in B} \text{bound}(x_i) \cup \bigcup_{x_i \in V - B} \text{unbound}(x_i)
\end{aligned}$$

It is easy to show that configurations always have this form in the DU algorithm. The set V consists of all variables in A and Σ. The set $B \subseteq V$ contains all the *bound* variables. The set $B_s \subseteq B$ contains all the bound variables whose binding is known on site s. The set $I \subseteq V$ contains all the variables whose binding has been *initiated* on some site but whose binding (if it exists) is not yet known on that site. The set $I_s \subseteq I$ contains all the variables whose binding has been initiated on site s but is not yet known on that site. In terms of the bind rules of Section 8.6.4, B corresponds to those variables for which the WIN rule has reduced. I corresponds to those variables for which the INITIATE rule has reduced but the corresponding ARRIVE rule has not yet reduced. Figure 8.15 illustrates the relationship between these five sets.

Two utility functions are used in the algorithm definition:

$$\begin{aligned}
\text{lhs}(\Sigma_s) &= \{x | \exists u.(x \leftarrow u)_s \in \Sigma_s \vee (x \leftarrow \bot)_s \in \Sigma_s\} \\
\text{var}(\Sigma_s) &= \text{lhs}(\Sigma_s) \cup \{x | \exists y, u.((y \leftarrow u)_s \in \Sigma_s \wedge u \equiv f(..., x, ...))\}
\end{aligned}$$

The function lhs(Σ_s) returns all bound and initiated variables of Σ_s. It generalizes the function lhs(σ) defined in Section 8.4.2. The function var(Σ_s) returns all variables mentioned in Σ_s, even if not bound.

Initial configuration

The initial configuration is $(A^{\text{init}};\Sigma^{\text{init}};\emptyset)$, with initial actions A^{init} that are all equations and $\Sigma^{\text{init}} = \{\text{unbound}(x_i) \mid x_i \in V\}$. We have initially $B = \emptyset$ and $I = \emptyset$.

Same as centralized setting	
true	Null action
false_s	Failure notification on site s
$(u = v)_s$	Equation on site s
New for distributed setting	
$x \sim u$	Binding request
$(x \Leftarrow u)_s$	Binding in transit to site s

Table 8.1: Actions in distributed configurations

Action

An action A is a multiset containing two new primitive actions in addition to the three primitive actions of the centralized setting (see Table 8.1). The new actions are needed to implement the distributed operations of the algorithm. The exact meaning of these actions is defined by the reduction rules that manipulate them. Intuitively, the action $x \sim u$ represents a message requesting the binding of x to u. For a given x, exactly one such action will cause a binding to be made; all others are discarded. The algorithm does not specify where the binding decision is made. An actual implementation can make the decision in a centralized or distributed way. In the Mozart implementation, the decision is centralized; it is made on the site where the variable was initially declared (see Section 8.9). The action $(x \Leftarrow u)_s$ represents a message containing a binding to x that may eventually become visible in Σ_s. As long as x is not in $\text{var}(\Sigma_s)$ then the binding stays "in the network".

Memo table

The global memo table M is the juxtaposition of all the local memo tables. That is, $M = \{(x = y)_s | x = y \in M_s\}$. Each local memo table M_s is a set of variable-variable equalities that has identical structure to the centralized memo table μ.

8.6.3 An example

Figure 8.16 gives an example execution that does distributed variable elimination. In this figure, thin solid arrows represent actions or bindings. Vertical bars "|" denote rule reductions, which happen in the numbered order shown. Thin dotted arrows represent causal links.

Initially, site 1 has equation $(x = a)_1$ and site 2 has equation $(x = b)_2$. Both sites do an INITIATE rule, which puts binding initiations $(x \leftarrow \bot)_s$ in both local stores. This ensures that the two equations cannot reduce until the binding arrives. We say that the equations are *suspended*. Equation $(x = b)_2$ is the first to do a WIN rule, and b therefore becomes the global binding of x. The other equation is discarded by the LOSE rule. The binding $(x \Leftarrow b)$ is sent to all sites. It arrives at each site through the ARRIVE rule. At that point, the suspended equations $(x = a)_1$ and $(x = b)_2$ become reducible again. The equation $(x = a)_1$ will cause an inconsistency to be flagged on site 1.

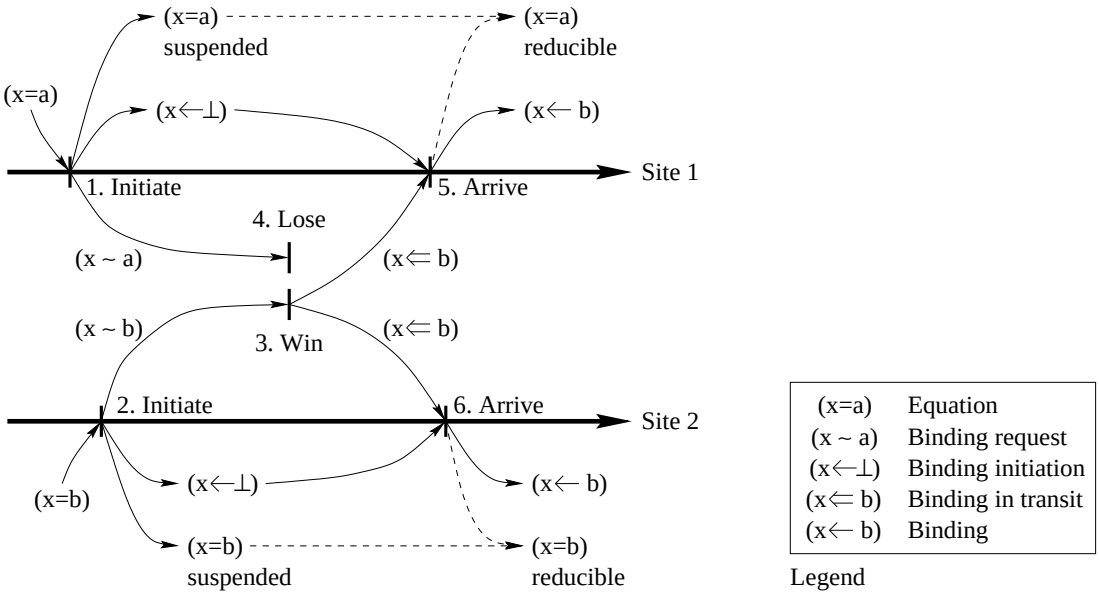


Figure 8.16: Distributed unification with $(x = a)_1$ and $(x = b)_2$

8.6.4 Definition

The DU algorithm has a close relationship to the CU algorithm. The structure and congruence rules continue to hold in the distributed setting. The only centralized rule that is changed is the BIND rule; it is replaced by the four bind rules below. It is clear from inspection that all six non-bind DU rules have identical behavior to the corresponding CU rules, if the CU rules are considered as acting on a single site.

Non-bind rules

These rules correspond exactly to the non-bind rules of the centralized algorithm. An inconsistency is flagged on the site that causes it by the action false_s.

$$\text{INTERCHANGE}\ \frac{(u=x)_s \parallel (x=u)_s}{\Sigma;\ M \parallel \Sigma;\ M}\quad \text{less}(u,x)$$

$$\text{MEMO}\ \frac{(x=u)_s \parallel \text{true}}{(x\leftarrow v)_s,\Sigma;\ M_s\cup M \parallel (x\leftarrow v)_s,\Sigma;\ M_s\cup M}\quad \begin{array}{l}\text{less}(u,x),\\ MEM(x=u,M_s)\end{array}$$

$$\text{DEREFERENCE}\ \frac{(x=u)_s \parallel (v=u)_s}{(x\leftarrow v)_s,\Sigma;\ M_s\cup M \parallel (x\leftarrow v)_s,\Sigma;\ ADD(x=u,M_s)\cup M}\quad \begin{array}{l}\text{less}(u,x),\\ \neg MEM(x=u,M_s)\end{array}$$

$$\text{IDENTIFY}\ \frac{(x=x)_s \parallel \text{true}}{\Sigma;\ M \parallel \Sigma;\ M}$$

$$\textsc{Conflict}\quad \frac{(t_1 = t_2)_s \;\|\; \text{false}_s}{\Sigma;\, M \;\|\; \Sigma;\, M} \quad t_1 = f_1(x_1,...,x_m),\, t_2 = f_2(y_1,...,y_n),\; (f_1 \neq f_2 \vee m \neq n)$$

$$\textsc{Decompose}\quad \frac{(t_1 = t_2)_s \;\|\; \bigwedge_{1\leq i\leq n}(x_i = y_i)_s}{\Sigma;\, M \;\|\; \Sigma;\, M} \quad t_1 = f(x_1,...,x_n),\; t_2 = f(y_1,...,y_n)$$

Bind rules

These rules replace the BIND rule of the centralized algorithm. The binding initiation $(x \leftarrow \bot)_s$ and the condition $x \notin \text{lhs}(\Sigma_s)$ in the INITIATE rule together ensure that only one binding attempt can be made per site.

$$\textsc{Initiate}\quad \frac{(x = u)_s \;\|\; x \sim u \wedge (x = u)_s}{\Sigma;\, M \;\|\; (x \leftarrow \bot)_s, \Sigma;\, M} \quad \text{less}(u,x),\; x \notin \text{lhs}(\Sigma_s)$$

$$\textsc{Win}\quad \frac{x \sim u \;\|\; \bigwedge_{s\in S}(x \Leftarrow u)_s}{\text{unbound}(x), \Sigma;\, M \;\|\; \text{bound}(x), \Sigma;\, M}$$

$$\textsc{Lose}\quad \frac{x \sim u \;\|\; \text{true}}{\text{bound}(x), \Sigma;\, M \;\|\; \text{bound}(x), \Sigma;\, M}$$

$$\textsc{Arrive}\quad \frac{(x \Leftarrow u)_s \;\|\; \text{true}}{\Sigma;\, M \;\|\; (x \leftarrow u)_s, \Sigma - \{(x \leftarrow \bot)_s\};\, M} \quad x \in \text{var}(\Sigma_s)$$

8.6.5 Dereference chains

A *dereference chain* in a store σ is a sequence of bindings $x_1 \leftarrow x_2$, ..., $x_{n-1} \leftarrow u_n$ with $n \geq 1$ and u_n unbound. We say the value of x_1 in store σ is u_n. To find u_n given x_1, it is necessary to follow the chain. A major difference between CU and DU is that CU always constructs dereference chains, whereas DU with eager variables forbids dereference chains to cross sites. Instead, DU copies remote terms to make them local. In a centralized setting, pointer dereferencing is fast, so the penalty of using dereference chains is small. This makes sharing terms very cheap. In a distributed setting, pointer dereferencing across sites is slow and it makes the current site dependent on the other site. This makes copying terms preferable.

Copying terms instead of creating dereference chains introduces redundant work. It is possible to reduce this work at the cost of more network operations. For example one can eagerly bind to variables and lazily bind to (big) nonvariable terms. This guarantees that a cross-site dereference chain has a maximum length of one.

DU with lazy variables allows dereference chains to cross sites. When the value is needed, the binding is requested from the owner. If the binding is another variable, then the process is repeated. Each iteration of this process corresponds to a dereference operation.

[123] presents a centralized binding algorithm that avoids all dereference chains. Variables that are bound together are put into a circular linked list. When one of them is bound to a value, the list is traversed and all members are bound. This makes accessing a variable's value a constant-time operation, at the price of making binding

	Distributed	Centralized
Action	true	true
	false_s	false
	$(u = v)_s$	$u = v$
	$x \sim u$	true
	$(x \Leftarrow u)_s$	$x \leftarrow u$
Store	$\text{bound}(x)$	true
	$\text{unbound}(x)$	true
	$(x \leftarrow \bot)_s$	true
	$(x \leftarrow u)_s$	$x \leftarrow u$

Table 8.2: Mapping from distributed to centralized configurations

more expensive. Taylor finds no significant performance difference between this algorithm and the standard algorithm when both are embedded in a Prolog system whose performance is comparable to a good C implementation.

8.7 Off-line total correctness

This section proves that the DU algorithm behaves as expected. We first define a mapping from any distributed to a centralized execution. Then we define a modification of the CU algorithm, the RCU algorithm, that models the redundant work done by the distributed algorithm. Then we prove safety and liveness properties by reducing the DU algorithm to the RCU algorithm. From this we show that the DU algorithm is correct.

We distinguish between the off-line total correctness and the on-line total correctness. In the off-line case, we have to show that the distributed algorithm terminates and gives correct results for any placement of a fixed set of initial equations. This can be done without any fairness assumptions. This is not true for the on-line case, which is handled in Section 8.8.

8.7.1 Mapping from distributed to centralized executions

The proofs in this section are based on a mapping m from any distributed configuration $(A;\Sigma;M)$ to a corresponding centralized configuration $(\alpha;\sigma;\mu)$. Distributed executions are mapped to centralized executions by mapping each of their configurations. The mapping m was designed according to the reasoning of Section 8.6.1. We show that m has very strong properties that lead directly to a proof that the distributed algorithm implements the centralized algorithm.

A primitive action is mapped to either a primitive action or a binding. A binding is always mapped to a binding. Other store contents map to true. In this way we can map any distributed configuration to a centralized one:

$$(A;\Sigma;M) \xrightarrow{m} (\alpha;\sigma;\mu) = (m_a(A);m_s(A,\Sigma);m_m(M))$$

Table 8.2 defines the mappings m_a and m_s for primitive actions and store contents. The mapping for all of A and Σ is the union of these mappings for all primitive actions in A and all store contents in Σ. The centralized memo table is the union of the tables on each site, namely $m_m(M) = \bigcup_{s \in S} M_s$.

The following diagram relates a distributed execution e and its corresponding centralized execution $m(e)$:

$$\begin{array}{ccc} (A;\Sigma;M) & \xrightarrow{e} & (A';\Sigma';M') \\ m \downarrow & & \downarrow m \\ (\alpha;\sigma;\mu) & \xrightarrow{m(e)} & (\alpha';\sigma';\mu') \end{array}$$

To show total correctness, i.e., that the distributed algorithm is an implementation of unification, we need to show both safety and liveness properties. A sufficient safety property is proved in Section 8.7.3: given any distributed execution e, the corresponding $m(e)$ is a correct centralized execution. A sufficient liveness property is proved in Section 8.7.4: given any e, its execution eventually makes progress. That is, if the last configuration of $m(e)$ is nonterminal, then the last configuration of e is nonterminal and continuing e will always eventually advance $m(e)$. In the distributed execution, the non-bind rules and the WIN rule are called *progressing* rules since they advance the centralized execution (see Table 8.3). The other rules are called *non-progressing*.

8.7.2 Redundancy in distributed unification (RCU algorithm)

This section defines and justifies a revised version of the CU algorithm, the RCU algorithm, that models the redundant work introduced by distributing rational tree unification. There are two sources of redundant work in the DU algorithm. The first source is due to the decoupling of binding initiation from binding arrival. A binding initiation for $(x = u)_s$ inhibits reduction of all equations of the form $(x = v)_s$. When a binding arrives on a site, these reductions become possible again, including the reduction of the original equation $(x = u)_s$. To make the original equation disappear, several rule reductions are needed including a DEREFERENCE, one or more IDENTIFY, and possibly a DECOMPOSE. This redundant work can be avoided in the implementation (see Section 8.9.5).

The second source of redundant work cannot be avoided in the implementation. It is due to each site having its own local memo table. Memo tables are needed because of rational trees with cycles. However, they are in fact a general caching technique that avoids doing any unification more than once. In the distributed algorithm, the information stored in each site's memo table is not seen by the other sites. Therefore each site has to reconstruct the part of the centralized memo table that it needs.

To model the local memo tables it suffices to weaken the memo table membership check. This affects the two rules MEMO and DEREFERENCE. Assume there are k sites. We introduce a weaker membership check MEM_k that is true only if the equation has been entered at least k times. This is implemented by extending the memo table to store pairs of an equation and the number of times the equation has been entered:

$$
\begin{array}{lcl}
ADD(x=y,\mu) & = & \mu \cup \{(x=y,1)\} \text{ if } (x=y,\_) \notin \mu \\
ADD(x=y,\mu) & = & \mu - \{(x=y,i)\} \cup \{(x=y,i+1)\} \text{ if } (x=y,i) \in \mu \\
ADD(x=t,\mu) & = & \mu \\
MEM(x=u,\mu) & = & \text{true if } (x=u,\_) \in \mu \\
MEM(x=u,\mu) & = & \text{false otherwise} \\
MEM_k(x=u,\mu) & = & \text{true if } (x=u,i) \in \mu \wedge i \geq k \\
MEM_k(x=u,\mu) & = & \text{false otherwise}
\end{array}
$$

The R-MEMO rule uses the new definition of *MEM*. The R-DEREFERENCE rule uses MEM_k and the new definition of *ADD*. If an equation has been entered from 1 to $k-1$ times then both rules are applicable. This is an example of using nondeterminism to model distributed behavior in a centralized setting.

Now we can update the CU algorithm to model the two sources of redundant work. We model memo table redundancy by replacing MEMO and DEREFERENCE by R-MEMO and R-DEREFERENCE. We model bind redundancy by replacing BIND by R-BIND, as defined below. The three new rules are as follows:

$$\text{R-BIND} \quad \frac{x=u \;\|\; x=u}{\sigma;\ \mu \;\|\; x \leftarrow u, \sigma;\ \mu} \quad \text{less}(u,x), x \notin \text{lhs}(\sigma)$$

$$\text{R-MEMO} \quad \frac{x=u \;\|\; \text{true}}{x \leftarrow v, \sigma;\ \mu \;\|\; x \leftarrow v, \sigma;\ \mu} \quad \begin{array}{l}\text{less}(u,x), \\ MEM(x=u,\mu)\end{array}$$

$$\text{R-DEREFERENCE} \quad \frac{x=u \;\|\; v=u}{x \leftarrow v, \sigma;\ \mu \;\|\; x \leftarrow v, \sigma;\ ADD(x=u,\mu)} \quad \begin{array}{l}\text{less}(u,x), \\ \neg MEM_k(x=u,\mu)\end{array}$$

Theorem 8.7.1 ((RCU total correctness)). *Given any initial configuration. Then the following two statements hold:*

1. *The RCU algorithm terminates.*
2. *All terminal configurations of the RCU and CU algorithms are logically equivalent to each other according to the definition of Section 8.5.2.*

Proof. We handle termination and correctness separately.

1. We know that CU terminates. The redundant work introduced by RCU has the following effect:
   - **Bind redundancy**. The R-BIND rule introduces extra rule reductions. The number of extra reductions is 2 if u is a variable and $2+a$ if u is a nonvariable and a is its arity.
   - **Memo table redundancy.** The memo table size for RCU is at most k times that of CU, which is finite. Hence only a finite number of extra rule reductions can be done.
2. For both bind and memo table redundancy, the additional equations are always duplicates of existing equations or equations of some previous configuration.

Distributed rule	Centralized rule
MEMO	R-MEMO
DEREFERENCE	R-DEREFERENCE
INTERCHANGE	INTERCHANGE
IDENTIFY	IDENTIFY
CONFLICT	CONFLICT
DECOMPOSE	DECOMPOSE
INITIATE	SKIP
WIN	R-BIND
LOSE	SKIP
ARRIVE	SKIP

Table 8.3: Correspondence between distributed and centralized rules

Therefore they add no additional information and the Entailment property still holds.

This completes the proof. □

8.7.3 Safety

Theorem 8.7.2 ((DU safety)). *If e is any execution of the DU algorithm, then $m(e)$ is an execution of the RCU algorithm, and the sequence of rules reduced in $m(e)$ can be constructed from e.*

Proof. We will prove that Table 8.3 correctly gives the centralized rule of $m(e)$ corresponding to a distributed rule in e. A "SKIP" rule means that no rule is executed. The proof is by induction on the length of execution e. In the base case, the initial configuration c_1 of e has an empty store and memo table, and a set of equations placed on different sites. Therefore $m(c_1)$ is a valid initial configuration for the centralized algorithm.

In the induction case, we assume that the theorem holds for an execution e. We need to show that for each distributed rule applicable in the last configuration of e, that doing this rule maps to doing a corresponding centralized rule. We do a case analysis over the distributed rules. Section 8.7.3 covers the non-bind rules and Section 8.7.3 covers the bind rules.

Non-bind rules

Decompose Assume that the distributed execution reduces a DECOMPOSE rule. Mapping the before and after configurations of the decomposition gives the following dia-

gram:

$$\begin{array}{ccc} \dfrac{(t_1 = t_2)_s \wedge A}{\Sigma; M} & \xrightarrow{DEC} & \dfrac{\bigwedge_i (x_i = y_i)_s \wedge A}{\Sigma; M} \\ m \downarrow & & \downarrow m \\ \dfrac{t_1 = t_2 \wedge m_a(A)}{m_s(A,\Sigma); m_m(M)} & \xrightarrow{X} & \dfrac{\bigwedge_i x_i = y_i \wedge m_a(A)}{m_s(A,\Sigma); m_m(M)} \end{array}$$

It is clear from inspection that rule X is a centralized decomposition.

Interchange, Identify, Conflict These three rules are handled in the same way as the DECOMPOSE rule.

Memo It is clear that the MEMO rule maps correctly to an R-MEMO rule, since from $M_s \subseteq \mu$ it follows that that $MEM(x = u, M_s) \Rightarrow MEM(x = u, \mu)$.

Dereference We now show that the DEREFERENCE rule maps to an R-DEREFERENCE rule. We have the following diagram (where $\Sigma_x = (x \leftarrow v)_s, \Sigma$):

$$\begin{array}{ccc} \dfrac{(x = u)_s \wedge A}{\Sigma_x; M_s \cup M} & \xrightarrow{DRF} & \dfrac{(v = u)_s \wedge A}{\Sigma_x; ADD(x = u, M_s) \cup M} \\ m \downarrow & & \downarrow m \\ \dfrac{x = u \wedge m_a(A)}{m_s(A,\Sigma_x); m_m(M_s \cup M)} & \xrightarrow{X} & \dfrac{v = u \wedge m_a(A)}{m_s(A,\Sigma_x); m_m(ADD(x = u, M_s) \cup M)} \end{array}$$

We know $\text{less}(u,x) \wedge \neg MEM(x = u, M_s)$. Since each site has its own local memo table, there can be only one redundant equation per site. Therefore $\neg MEM(x = u, M_s)$ implies that $\neg MEM_k(x = u, \mu)$. That is, if at least one local memo table does not contain $x = u$, then the centralized memo table contains $x = u$ less than k times. It follows that rule X is exactly an R-DEREFERENCE rule. This shows that relaxing the memoization condition to do only a check on the local part of the memo table is correct but may introduce one redundant equation per site.

Bind rules

Initiate

$$\begin{array}{ccc} \dfrac{(x = u)_s \wedge A}{\Sigma; M} & \xrightarrow{INI} & \dfrac{x \sim u \wedge (x = u)_s \wedge A}{(x \leftarrow \bot)_s \Sigma; M} \\ m \downarrow & & \downarrow m \\ \dfrac{x = u \wedge m_a(A)}{m_s(A,\Sigma); m_m(M)} & \xrightarrow{X} & \dfrac{x = u \wedge m_a(A)}{m_s(A,\Sigma); m_m(M)} \end{array}$$

It is clear from inspection that X is a SKIP rule. After the SKIP, we know that $\text{less}(u,x)$.

Win

$$\frac{x \sim u \wedge A}{\text{unbound}(x), \Sigma; M} \xrightarrow{WIN} \frac{\bigwedge_{s \in S}(x \Leftarrow u)_s \wedge A}{\text{bound}(x), \Sigma; M}$$

$$\frac{m \downarrow \quad m_a(A)}{m_s(A, \Sigma); m_m(M)} \xrightarrow{X} \frac{\downarrow m \quad m_a(A)}{(x \leftarrow u), m_s(A, \Sigma); m_m(M)}$$

It is not immediately clear that transition X maps to a rule. We will show that X maps to an R-BIND rule. First, we show that $x = u$ is in $m_a(A)$. From $x \sim u$ we know that an INITIATE has been done and unbound(x) means no WIN has yet been done; together this means that A contains $(x = u)_s$, which maps to $x = u$. Second, less(u, x) because the INITIATE requires this and its truth is never altered. Third, $x \notin \text{lhs}(m_s(A, \Sigma))$ since no WIN has been done and the only way to get a centralized binding for x is through a WIN. Taken together, these three statements imply that the centralized transition is an R-BIND reduction.

Lose, Arrive These rules trivially map to a SKIP rule.

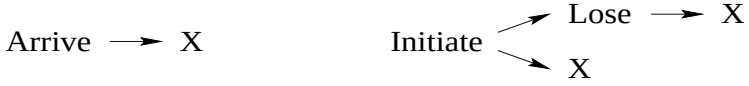
This proves the theorem. □

8.7.4 Liveness

It remains to show that the DU algorithm always terminates in a configuration that maps to a terminal configuration of the RCU algorithm. The main step is to show that the DU algorithm always "makes progress", in the sense of this section. As a corollary, it follows that the DU algorithm is deadlock-free. We first prove a small lemma about the non-progressing rules.

Lemma 1 ((Finiteness of non-progressing DU execution)). *Given any valid configuration d of the DU algorithm, then the number of consecutive non-progressing rules that can be reduced starting from d is finite. The resulting configuration d' satisfies $m(d') = m(d)$.*

Proof. The proof is by induction on the length of the execution e of which d is the last configuration. We assume that the lemma holds for all configurations of e before d. We show that it holds for d by enumerating all possible executions of non-progressing rules. Consider all rules that manipulate actions based on the same variable-term pair (say, x and u). Denote a configuration in which a non-progressing rule is applicable by the name of that rule. Denote by X a configuration in which no rules or only progressing rules are applicable. By inspecting the rules, we deduce the following graph of causal relationships:



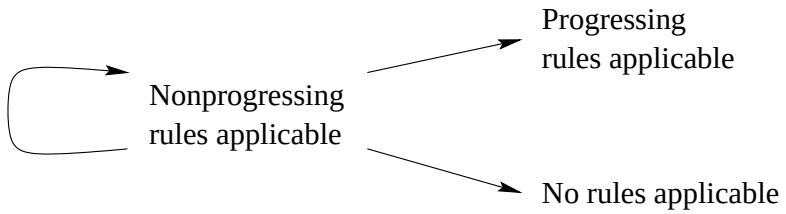


Figure 8.17: Non-progressing transitions in the DU algorithm

That is, applying INITIATE possibly leads to a configuration in which LOSE is applicable, and so forth. A graph identical to this one exists for all variable-term pairs. In all resulting sequences there are no cycles. Therefore a configuration in which some non-progressing rules are applicable will eventually lead to one in which no non-progressing rules are applicable. □

Theorem 8.7.3 ((DU liveness)). *Assume that e is any execution of the DU algorithm such that the last configuration of $m(e)$ is nonterminal in the RCU algorithm. Then the last configuration of e is nonterminal and any execution that has e as the initial segment will eventually reduce a progressing rule in its continuation beyond e.*

Proof. Assume a distributed execution e with last configuration d_i such that $c = m(d_i)$ is nonterminal. We must show that $\exists j > i : d_i \to \cdots \to d_{j-1} \to d_j$ where $d_{j-1} \to d_j$ is an application of a progressing rule. Any execution starting from d_i and doing non-progressing rules as long as possible must initially follow the state diagram of Figure 8.17. Applying the lemma, we can assume that no non-progressing rules are applicable in d_{j-1}. It remains to show that a progressing rule is always applicable there. We do a case analysis over the RCU rules. Let the RCU configuration be c, so that $m(d_{j-1}) = c$. For each rule, we apply the inverse of mapping m, and we attempt to infer whether a progressing rule is applicable.

Interchange

Assume that the INTERCHANGE rule is applicable in c. Therefore $\text{less}(u,x)$ holds and c contains $u = x$. For some site s, d_{j-1} contains $(u = x)_s$. Therefore the INTERCHANGE rule is applicable in d_{j-1}.

R-Memo, R-Dereference

Except for the memo table, the conditions for these two rules are identical. For both R-MEMO and R-DEREFERENCE, c contains $x = u$ and $x \leftarrow v$ and we know $\text{less}(u,x)$, Therefore for some site s, d_{j-1} contains $(x = u)_s$. For this site, d_{j-1} contains one of $(x \leftarrow v)_s$ or $(x \Leftarrow v)_s$. The case $(x \Leftarrow v)_s$ is impossible by the lemma, since in that case one of ARRIVE or INITIATE is applicable depending on whether or not $(x \leftarrow \perp)_s$ is in

the store. If $MEM(x = u, M_s)$ then a MEMO is applicable. Otherwise, a DEREFERENCE is applicable.

R-Bind

Both less(u,x) and $x \notin \text{lhs}(\sigma)$ hold. For some site s, d_{j-1} contains $(x = u)_s$. This site must also contain $(x \leftarrow \perp)_s$, since otherwise an INITIATE is applicable. Since $x \notin \text{lhs}(\sigma)$, we know unbound$(x)$, so the $x \sim u$ of the INITIATE still exists and a WIN rule is applicable.

Identify, Conflict, Decompose

These are straightforward.

This proves the theorem. □

8.7.5 Total correctness

Theorem 8.7.4 ((DU total correctness)). *Given any finite multiset of equations, then placing them on arbitrary sites and executing the DU algorithm will terminate and result in a configuration that maps to a configuration equivalent to that of a terminating CU execution.*

Proof. From DU safety, any results obtained are correct results for the RCU algorithm. From DU liveness and the Finiteness Lemma, the DU algorithm will terminate and reach this correct result. From RCU total correctness, the result is equivalent to the result of a terminating CU execution. □

8.8 On-line total correctness

In the real system, it is almost never the case that unification is initiated with a fixed set of equations and runs to termination without any interaction with the external environment. Rather, the algorithm will be running indefinitely, and from time to time an equation will be added to the current action. This is the *on-line* case. The interesting property is not termination, but whether the equation will be entailed by the store in a finite number of reductions (finite entailment). In this section, we extend the CU and DU algorithms to the on-line case, and we show that the extended algorithms satisfy the finite entailment property. We use the standard weak fairness assumption that any rule instance applicable infinitely often will eventually be reduced. We show that this is not enough to guarantee that the equation will be entailed, but that we need an additional property, the finite size property, to bound the amount of work needed to incorporate the equation in the store.

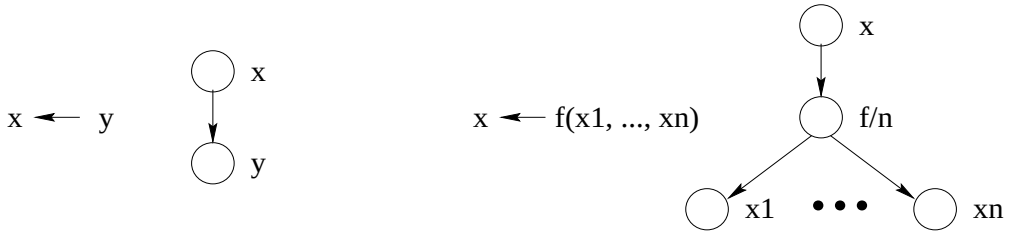


Figure 8.18: Mapping the store to its graph

8.8.1 On-line CU and DU algorithms

We extend the CU algorithm (from now on called the *off-line* CU algorithm) with a new rule:

$$\textsc{Introduce} \quad \frac{\text{true} \parallel u = v}{\sigma;\ \mu \parallel \sigma;\ \mu}$$

This rule is always applicable and adds a new equation to the action when it reduces. The extended algorithm, also called the *on-line* CU algorithm, therefore does not terminate. We extend the DU algorithm in a similar way by an INTRODUCE rule that introduces $(u = v)_s$ for an arbitrary site s.

8.8.2 Finite size property

Any store σ can be mapped to a graph with two kinds of nodes, variables and records. The graph is defined in a straightforward way from the store's bindings (see Figure 8.18):

- A binding $x \leftarrow y$ maps to variable nodes x and y, with a directed edge from x to y.
- A binding $x \leftarrow f(x_1,\ldots,x_n)$ maps to a set of variable nodes x, x_1, ..., x_n and a record node f/n, with directed edges from x to f/n and from f/n to every x_1, ..., x_n.

Given a variable node x, we define $\text{graph}(x,\sigma)$ as the subgraph of σ's graph whose nodes and edges are reachable from x. We also define $\text{size}(x,\sigma)$ as the number of edges in this subgraph. This quantifies the size of the data structure attached to x.

Given a valid configuration with store σ, it is clear that $\text{size}(x,\sigma)$ is finite. However, the size may be unbounded when considering all configurations of a given execution. This leads us to define the following property. We say a variable x has the *finite size* property for the execution e if:

$$\exists n \geq 0 : \forall (\_;\sigma_k;\_) \in e : \text{size}(x,\sigma_k) \leq n$$

That is, there is a finite upper bound on the size of x that holds for the whole execution. We say that an equation $u = v$ has the finite size property if all its variables do. The finite size property is used to avoid infinite executions caused by race conditions in two cases:

1. Dereference chains that increase in length indefinitely. For example, consider the equation $x_0 = y_0$, which is accompanied by the infinite sequence of pairs of equations $x_i = x_{i+1}$ and $y_i = y_{i+1}$, starting with $i = 0$. These equations are added by an INTRODUCE rule at the appropriate times. We assume that the ordering condition enforces that lower-indexed variables are bound to higher-indexed variables. For each i starting with 0, if $x_i = x_{i+1}$ and $y_i = y_{i+1}$ are both introduced and bound before $x_i = y_i$ is dereferenced, then the store will never entail $x_0 = y_0$.

2. Nested terms that increase in depth indefinitely. For example, consider the equation $x_0 = y_0$, which is accompanied by the equations $x_i = f(x_{i+1})$ and $y_i = f(y_{i+1})$, starting with $i = 0$. For each i starting with 0, if $x_i = f(x_{i+1})$ and $y_i = f(y_{i+1})$ are both introduced and bound before $x_i = y_i$ is decomposed, then the store will never entail $x_0 = y_0$.

It is remarkable that these two infinite executions are possible even with the weak fairness assumption. One way to avoid infinite executions would be to give the INTRODUCE rule lower priority than the others, i.e., as long as another rule is applicable, do not reduce an INTRODUCE rule. But this does not model the real world, in which equations can arrive at any time. The finite size property does not have this deficiency. It does not restrict in any way *when* new equations are introduced. Rather, it forbids introducing any sequence of equations that would cause a problem.

The finite size property can be enforced easily for dereference chains by requiring that all new variables have higher order than all existing variables. Then the total length of all dereference chains that need traversing is bounded by the number of variables in the system when the equation is introduced.

In the case of nested structures, the finite size property can be enforced by avoiding to unify terms whose nesting depth is potentially unbounded. This seems to be a reasonable condition because when a potentially infinite unification is necessary in practice, then it is sufficient that it always makes progress, not that it completes (see, e.g., the streams of Section 8.3.4). The weak fairness assumption is enough by itself to guarantee progress of infinite unifications and eventual termination of finite unifications. The finite size property ensures that a unification that is intended by the programmer to be finite will actually be finite during the execution. These two conditions suffice for all practical programs we know of.

8.8.3 Total correctness

Under what conditions will the store entail a given equation after a finite number of reductions? First, there must be no detected inconsistencies (false actions) within the

context of the given memo table. Second, the amount of work needed to incorporate the equation into the store must be finite (finite site property).

An inconsistency is detected at most once per memo table. This is true for both the centralized and distributed algorithms as well as the Mozart implementation. In the CU algorithm, there is only one memo table, so an inconsistency is detected at most once. In the DU algorithm, there is a memo table per site, so an inconsistency can be detected once per site.

Theorem 8.8.1 ((Finite entailment of on-line CU)). *Given any valid configuration c of the on-line CU algorithm that contains the equation $u = v$. Given any execution e that contains c and satisfies the finite size property for $u = v$. Then e will eventually contain either a false action or a store that entails $u = v$.*

Proof. We are given that size$(u = v, \sigma_k)$ has a finite upper bound in e. Therefore graph$(u = v, \sigma_k)$ has a finite limit graph. Let V denote the set of variables in this graph. Denote the store corresponding to the limit graph as σ_V. Since V has a finite limit, the set $\mu_V = \{x = y \in \mu_k | x, y \in V\}$, i.e., of equalities in μ_k whose variables are in V, also has a finite limit. When this limit is reached, then consider the equations α_V, part of α_k, whose variables are all in V. Consider an execution starting with $(\alpha_V; \mu_V; \sigma_V)$, without the INTRODUCE rule, and that reduces rules in the same order as e does. This is a continuation of an off-line CU execution. If no false actions occur, then the Entailment Property (see Section 8.5.2) implies that eventually we end up with a store that entails $u = v$. □

We now extend this result to the distributed case. First we extend the DU algorithm to an on-line DU algorithm by an INTRODUCE rule that introduces an equation on any site. It is easy to see that safety continues to hold. We now show liveness and finite entailment for the on-line DU algorithm.

Theorem 8.8.2 ((Liveness of on-line DU)). *Given weak fairness and any distributed execution e of the on-line DU algorithm such that $m(e)$ is nonterminal, then continuing e will always eventually reduce a progressing rule.*

Proof. Minor modification of the proof of DU liveness, using weak fairness to compensate for the INTRODUCE rule. □

Theorem 8.8.3 ((Finite entailment of on-line DU)). *Given any valid configuration d of the on-line DU algorithm that contains the equation $(u = v)_s$. Given any execution e that contains d and such that $m(e)$ satisfies the finite size property for $u = v$. Then e will eventually contain either a false$_s$ action or a store on site s that entails $u = v$.*

Proof. We outline the proof. The execution on site s has a local memo table M_s for site s. We consider this execution to be a centralized execution with memo table $\mu = M_s$. By the previous theorem, the result holds for the centralized execution. Therefore the result holds also for the distributed execution on site s. □

8.9 The Mozart implementation

The Mozart system contains a refined version of the on-line DU algorithm, called "Mozart algorithm" in what follows. Section 8.9.1 summarizes how the implementation differs with respect to the on-line DU algorithm. Section 8.9.2 introduces the distribution graph, which is the framework in which the Mozart algorithm is defined. Then Section 8.9.3 defines the properties of the network and the notation used to define the distributed algorithm. After these preliminaries, the algorithm itself is defined. Section 8.9.4 defines the local algorithm and Section 8.9.5 defines the distributed algorithm.

8.9.1 Differences with on-line DU

The Mozart algorithm refines the on-line DU algorithm by making concrete decisions regarding several aspects that were left open. Furthermore, the Mozart algorithm does several optimizations to improve performance and has several extensions including a model for failure detection and handling. This section summarizes these refinements, optimizations, and extensions.

Refinements

Separation into local and distributed algorithms The Mozart algorithm consists of two parts: a purely local algorithm (corresponding to the DU non-bind rules, see Section 8.9.4) and a distributed algorithm (corresponding to the DU bind rules, see Section 8.9.5). A thread wishing to tell an equation invokes the local algorithm. To bind a distributed variable to another one or to a record, the local algorithm invokes the distributed algorithm. The thread blocks, waiting for a reply. When the variable binding is known locally, then the thread continues.

The owner site Each distributed variable is managed from a special site, the owner site, which is where the variable was originally created. This site contains the variable's unbound/bound flag and other information, e.g., the register list (see below).

Variable ordering The Mozart algorithm implements the order relation $\text{less}(u,v)$ as follows. Records are less than distributed variables are less than local variables. Distributed variables are totally ordered, local variables are totally ordered per site, and records are unordered. Local variables are ordered according to a per-site index i that is incremented for each new variable.[5] Distributed variables are ordered according to a pair (s,i) where s is the site number on which the variable was initially created and i is the index of the variable on that site. From this ordering relation it follows that if the number of sites is finite and data structures with unbounded depth are not created, then the Mozart algorithm satisfies the finite size property (see Section 8.8.2).

[5] For local variables, the index is simply the variable's address.

Optimizations

Globalization The Mozart algorithm distinguishes between local and distributed variables (see Section 8.9.5).

Variable registration A variable binding is not sent to all sites, but only to registered sites (see Section 8.9.5).

Grouping nested data structures Binding a nested data structure to a distributed variable is done by the Mozart algorithm as a single operation (see Section 8.9.5).

Winner optimization When a variable is bound to a term, then the term does not have to be sent back to the site that initiated the binding (see Section 8.9.5).

Asynchronous streams To allow streams to be created asynchronously, variables are given a set of registered sites as soon as they are globalized (see Section 8.9.5).

Extensions

Lazy and eager variables The laziness property affects the moment when the variable is registered. Eager proxies are registered immediately. Lazy proxies delay registration until a binding attempt is made (see Section 8.9.5).

Read-only logic variables Standard logic variables have two operations, reading the value and binding. For security reasons, it is often useful to prohibit binding, for example, when passing the variable to a less-trusted site [92].

Garbage collection Distributed garbage collection is based on a credit mechanism that collects all garbage except cross-site cycles between stateful entities (see Section 8.9.2).

The failure model The Mozart algorithm is conservatively extended with a model for failure detection and handling that reflects network and site failures to the language level (see Section 8.9.2).

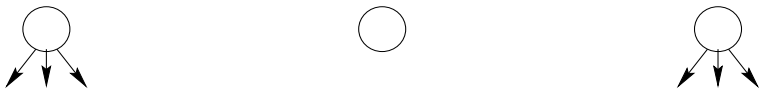


Figure 8.19: The three node types of the language graph

8.9.2 The distribution graph

We model distributed executions in a simple but precise manner using the concept of *distribution graph*. We obtain the distribution graph in two steps from an arbitrary execution state of the system. The first step is independent of distribution. We model the execution state by a directed graph, called *language graph*, in which a record, an unbound variable, and a thread each correspond to one node (see Figure 8.19). The edges in this graph denote the node's references: a record and a thread refer to other nodes; an unbound variable has no references.

In the second step, we distribute the execution over a set of sites. Assume a finite set of sites and annotate each node by its site (see Figure 8.20). If a variable node, e.g., `N2`, is referenced by at least one node on another site, then map it to a *set* of nodes, e.g., {`P1`,`P2`,`P3`,`M`}. This set is called the *access structure* of the original node. An access structure consists of one *proxy node* `Pi` for each site that referenced the original node and one *owner node* `M` for the whole structure. The resulting graph, containing both local nodes and access structures where necessary, is called the *distribution graph*. The execution of the distributed algorithm is defined in terms of this graph. A logic variable with an access structure is called a *distributed* variable (as opposed to a *local* variable). A variable referenced on more than one site certainly has an access structure.

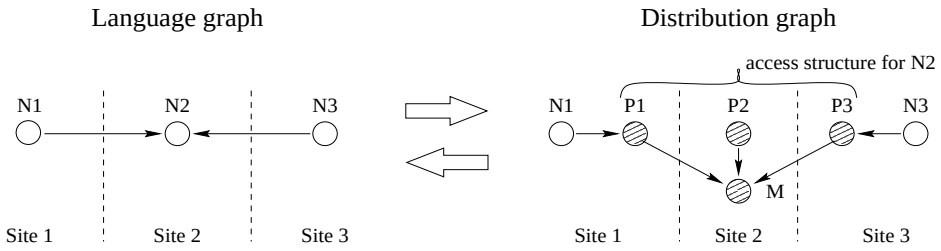


Figure 8.20: An access structure in the distribution graph

Each access structure is given a global name n that is unique system-wide. In Distributed Oz, n is the pair (s,i) that is also used to order the distributed variables. The global name n encodes (among other things) the owner site s. Furthermore, a proxy node is uniquely identified by the pair (n,s'), which contains the proxy's site s'. On each site, n indexes into a table that refers to the proxy. This allows to enforce the invariant that each site has at most one proxy.

Messages are sent between nodes in access structures. In terms of sites, a message is sent from the source node's site to the destination node's site. The message may contain a subgraph of the distribution graph. Just before the message leaves the source site, a new access structure is created for each local variable in the subgraph. The message refers only to proxy nodes, not to local variables. When the message arrives, the subgraph becomes known at the destination site. Each proxy node is looked up in the site table. If it is not present, then a new proxy node is created and entered in the

table. This extends an existing access structure with one new proxy. The process of creating or extending an access structure is called *globalization* (see Section 8.9.5).

The behavior of a distributed variable is defined as a protocol between the nodes of its access structure. In general, nodes other than variable nodes can also have access structures, and therefore obey a protocol. The Distributed Oz implementation uses four non-trivial protocols. Three are designed for specific language entities, namely variables, object records, and state pointers. Variables use a *variable binding* protocol, which is part of the distributed unification algorithm and is presented in this article. Object records use a *lazy replication* protocol. State pointers use a *mobile state* protocol. The fourth protocol is a distributed garbage collection algorithm using a credit mechanism. Garbage collection is part of the management of access structures, and it therefore underlies the other three protocols. See [7, 132, 131, 54] for more information on these protocols.

Distributed garbage collection

Distributed garbage collection is based on a credit mechanism, a variant of reference counting [99]. The credit mechanism interfaces with the local garbage collectors on each site. All distributed garbage is removed except for cross-site cycles between stateful entities.

The global name of an access structure is associated with a pool of *credits*. The owner site lends credit to sites and messages that refer to the global name. A owner site has an integer corresponding to the total number of credits lent. A proxy site must hold at least one credit for the access structure. The proxy site keeps a count of how many credits it has borrowed from the owner site. If local garbage collection removes the proxy node, then its credit is returned to the owner site. The global name can be reclaimed at the owner site when no more borrowed credits exist. The owner node itself can be reclaimed if it has both no global name and no local references. When that happens, the distributed variable becomes local again–we say it is *localized* (see Section 8.9.5).

The failure model

The failure model is designed according to the principle that a program should be able to make all decisions regarding failure behavior [130, 54, 19]. That is, the implementation does not make any irrevocable decisions by default. Full treatment of the model is beyond the scope of this paper. We briefly summarize the main ideas. The failure model considers failures at the level of individual language entities, e.g., logic variables. The model covers permanent site failures and temporary and permanent network failures. The model has two effects at the language level:

- It extends each operation on an entity to have three possible results. If there is no failure then the operation succeeds. If there is a failure, then the operation either waits indefinitely until the problem goes away or aborts and is replaced by a user-defined procedure call. The call can retry the operation. There are

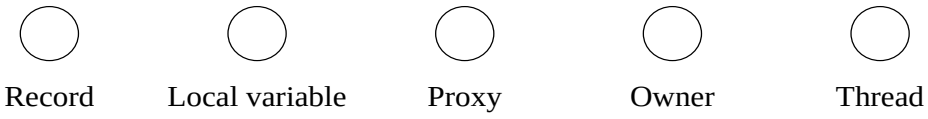


Figure 8.21: The five node types of the distribution graph and their message interfaces

no default time-outs; it is up to the program to decide whether to continue to wait or not. For example, an entity can be configured so that all operations wait indefinitely on network failures (in the hope that they are temporary) and raise an exception on a permanent site failure.

- It allows to eagerly detect a problem with an entity, i.e., without having to do an operation on the entity. When a user-specified failure condition is detected then a user-defined procedure is called in its own thread. The failure condition does not necessarily keep the entity from working, i.e., it can just give information. For example, a remote proxy site failure will often have no effect at all on the binding of a logic variable, but it may nonetheless be important to be notified of this failure.

This failure model is fully implemented as part of the Mozart system. We are currently developing more powerful abstractions in Oz on top of this model.

8.9.3 Basic concepts and notation

Network

Consider a single owner node M, a set of k proxy nodes P_i with $1 \leq i \leq k$, and a set of m thread nodes T_i with $1 \leq i \leq m$. All nodes have state and interact according to Figure 8.21. Thread, proxy, and owner nodes send messages to each other and perform internal operations. Record and local variable nodes interact only with thread nodes. Let these nodes be linked together by a network N that is a multiset containing messages of the form $d : m$ where d identifies a destination (proxy, owner, or thread node) and where m is a message.

The Mozart algorithm is defined using reduction rules of the form

$$\frac{\text{Condition}}{\text{Action}} \quad .$$

Each rule is defined in the context of a single node. Execution follows an interleaving model. The local algorithm imposes an order on how its rules are fired; see the pseudocode definition of Section 8.9.4. The distributed algorithm imposes no such order.

At each reduction step, a rule with valid condition is selected. Its associated actions are reduced atomically. A rule condition consists of boolean conditions on the node

Node	Attribute	Type
Any node	id	NodeId
	type	{RECORD,LOCVAR,PROXY,MANAGER,THREAD}
Record	label	Atom
(N.type=RECORD)	arity	Integer
	args	**array**[1..arity] **of** Node
Local variable	state	{UNBOUND, BOUND(Node)}
(N.type=LOCVAR)	eager	{FALSE, TRUE}
Proxy	state	{UNBOUND, INITIATED, BOUND(Node)}
(N.type=PROXY)	eager	{FALSE, TRUE}
	reg	{FALSE, TRUE}
	owner	NodeId
Owner	state	{UNBOUND, BOUND(Node)}
(N.type=MANAGER)	reglist	**set of** NodeId
Thread		
(N.type=THREAD)		

Table 8.4: Node state

state and one optional receive condition **Receive**(*d*,*m*). The condition **Receive**(*d*,*m*) means that $d : m$ has arrived at *d*. Executing a rule with a receive condition removes $d : m$ from the network and performs the action part of the rule. A rule action consists of a sequence of operations on the node state with optional sends. The action **Send**(*d*,*m*) asynchronously sends message *m* to node *d*, i.e., it adds the message $d : m$ to the network.

We assume that the network and the nodes are *fair* in the following sense. The network is asynchronous, and messages to a given node take arbitrary finite time and may arrive in arbitrary order. All rules that are applicable infinitely often will eventually reduce.

Node state

Table 8.4 defines the state of the five node types by listing the attributes of each node. All nodes have attributes "id" and "type", which have constant values. The types NodeId, Atom, and Integer are atomic types implemented in a straightforward way. In the real system, threads, proxies, and owners have more attributes, for example, threads have an execution state and proxies and owners maintain information for distributed garbage collection.

Utility operations

The memo table uses the function clearmemo(), the procedure add(N_1,N_2,M) and the boolean function mem(N_1,N_2,M). The latter two are exactly the *ADD* and *MEM* operations defined in Section 8.4.2. The other operations are defined as follows:

```
eq(N1,N2)          =  N1.id=N2.id
locvar(N)          =  N.type=LOCVAR
disvar(N)          =  N.type=PROXY
nonvar(N)          =  N.type=RECORD
var(N)             =  locvar(N) ∨ disvar(N)
bound(N)           =  if var(N) → N.state=BOUND(_) fi
initiated(N)       =  if disvar(N) → N.state=INITIATED fi
deref1(N)          =  if var(N) ∧ bound(N) →
                         N1 where N.state=BOUND(N1) fi
compatible(N1,N2)  =  if nonvar(N1) ∧ nonvar(N2) →
                         N1.arity=N2.arity ∧ N1.label=N2.label fi
proxyids(N)        =  if locvar(N) → {}
                      [] disvar(N) →
                         if bound(N) → proxyids(deref1(N))
                         [] not bound(N) → {N.id} fi
                      [] nonvar(N) → ⋃1≤i≤N.arity proxyids(N.args[i]) fi
```

8.9.4 The local algorithm

Figure 8.22 defines the local algorithm, which executes in each thread that does a unification. The definition follows closely the non-bind rules of Section 8.6.4, where a rule corresponds to a guard and its body. The two main differences are that the local algorithm maintains a memo table that is shared among the rules and that a sequential order is imposed on rule reductions. The **if** is a guarded command that suspends until at least one of its guards is true.

In the implementation, executions of the local and distributed algorithms are not interleaved. Rather, the local algorithm is executed atomically until it exits (either normally or through an exception) or until it blocks after sending a binding request.

Each invocation of unify is done within a thread. The unification completes in two ways: when it terminates normally or when an inconsistency is detected, in which case a failure exception is raised. During the unification, the thread will block when waiting for a binding request to complete. This works as follows. The first thread that tries to bind a variable will send a binding request (INITIATE rule). Other threads that try to bind the same variable on the same site will not send any message. All threads then block at the **if**: no guard is true because the variable satisfies disvar(N_1) ∧ **not** bound(N_1) ∧ initiated(N_1). As soon as the binding arrives, bound(N_1) is true and the **if** becomes reducible. All threads can then continue.

The local algorithm is optimized to bind local variables locally. With a suitable data representation, the algorithm is implementable very efficiently [48, 127]. The implementation does binding in place and dereferencing inline. Common cases of unification such as parameter passing and local variable initialization do not need binding nor dereferencing, but reduce to single register moves or stores.

The local memo table is implemented by means of *forwarding pointers* between variables [52]. That is, when the equation $x = y$ is encountered for the first time,

```
procedure unify(N1,N2)
   define memotable M
   define procedure inner_unify(N1,N2)
      if /**** INTERCHANGE ****/
         var(N2), less(N1,N2) → inner_unify(N2,N1)
      [] /**** IDENTIFY ****/
         var(N1), eq(N1,N2) → skip
      [] /**** MEMO ****/
         var(N1), less(N2,N1), bound(N1), mem(N1,N2,M) → skip
      [] /**** DEREFERENCE ****/
         var(N1), less(N2,N1), bound(N1), not mem(N1,N2,M) →
         add(N1,N2,M)
         inner_unify(deref1(N1),N2)
      [] /**** BIND ****/
         locvar(N1), less(N2,N1), not bound(N1) →
         N1.state ← BOUND(N2)
      [] /**** INITIATE ****/
         disvar(N1), less(N2,N1), not bound(N1), not initiated(N1) →
         N1.state ← INITIATED
         Send(N1.owner,binding_request(N2))
         clearmemo(M)
         inner_unify(N1,N2)
      [] /**** DECOMPOSE ****/
         nonvar(N1), nonvar(N2), compatible(N1,N2) →
         for i:=1 to N1.arity do inner_unify(N1.args[i],N2.args[i])
      [] /**** CONFLICT ****/
         nonvar(N1), nonvar(N2), not compatible(N1,N2) →
         raise failure_exception
      fi
   end
in
   clearmemo(M)
   inner_unify(N1,N2)
end
```

Figure 8.22: Distributed unification part 1: Local algorithm

/**** WIN ****/

Receive(M.id,binding_request(N)) $\wedge$ M.state=UNBOUND

$\forall i \in$ M.reglist: **Send**(i, binding_in_transit(N))
M.state $\leftarrow$ BOUND(N)

/**** LOSE ****/

Receive(M.id,binding_request(_)) $\wedge$ M.state=BOUND(_)
---
skip

/**** ARRIVE ****/

Receive(P.id,binding_in_transit(N)) $\wedge$ (P.state=UNBOUND $\vee$ P.state=INITIATED)

$\forall i \in$ proxyids(N): **Send**(i, reg)
P.state $\leftarrow$ BOUND(N)

/**** Variable registration ****/

Receive(P.id,reg) $\wedge$ P.reg=FALSE

P.reg $\leftarrow$ TRUE
Send(P.owner,register(P.id))

Receive(P.id,reg) $\wedge$ P.reg=TRUE
---
skip

Receive(M.id,register(PId)) $\wedge$ M.state=UNBOUND

M.reglist $\leftarrow$ M.reglist $\cup \{$PId$\}$

Receive(M.id,register(PId)) $\wedge$ M.state=BOUND(N)

Send(PId,binding_in_transit(N))

Figure 8.23: Distributed unification part 2: Distributed algorithm

a forwarding pointer is installed from x to y. This allows a very fast check of memo table membership. Namely, if $x = y$ or $y = x$ is encountered later on, then dereferencing will reduce the equation to $y = y$, which does no further work. The forwarding pointers are installed in the context of a single atomic unification operation. They are removed when the local algorithm exits or blocks.

Other operations can be performed on a site while a unification is blocked. For correctness, the forwarding pointers must be removed whenever execution leaves the local algorithm. This is modeled in Figure 8.22 by creating a new memo table when unify is called and by clearing the memo table after an INITIATE rule. This means that the memo table starts from empty at each atomic execution of the local algorithm. The Mozart algorithm therefore potentially does more redundant work than the on-line DU algorithm, because the DU algorithm never clears the local memo tables.

8.9.5 The distributed algorithm

Figure 8.23 defines the distributed algorithm, which extends the DU bind rules of Section 8.6.4 with globalization and variable registration. The implementation does three other important optimizations, namely grouping nested data structures, the winner optimization, and asynchronous streams. For clarity, we do not define the latter formally, but rather show how to extend the protocol to include them. We also explain how to extend the protocol for lazy and eager variables.

Globalization

Newly-created variables are always local. When a message is sent referencing a local variable, then a new distributed variable is created and the local variable is bound to it. This is called *globalizing* the local variable. An access structure is created when a local variable is globalized. When the message arrives then a new proxy will be created for the distributed variable if none exists on the arrival site. Therefore globalization is part of the **Send** and **Receive** operations [7]. The inverse operation, *localization*, consists of removing the access structure when the variable is only referenced on one site (see Section 8.9.2). The distributed variable becomes a local variable again.

All variables have a boolean attribute "eager" that determines whether the variable is eager or lazy. This attribute affects only the network operations of the distributed algorithm. Assume we have a local variable L with L.state=UNBOUND and L.eager=*b*. After globalizing, the original site contains three nodes, L, P, and M, with the following states:

L.state=BOUND(P)
P.state=UNBOUND, P.eager=*b*, P.reg=*b*, P.owner=M.id
M.state=UNBOUND, M.reglist=**if** *b* **then** {P.id} **else** {} **fi**

Variable registration

In the DU algorithm, a binding arrives on a site if the variable exists in the site's store. In the Mozart algorithm, the variable's owner keeps track of the sites that reference

the variable. A site that receives a distributed variable for the first time (i.e., when a term containing the variable first arrives on the site) has to register with the owner in order to receive the variable's binding. In Figure 8.23, first the ARRIVE rule reduces, which sends reg messages to all proxies in the binding. The reg message causes all unregistered proxies to register with their owner. When a variable is bound, then a binding_in_transit message is sent to all registered sites. If the variable is already bound when the register message arrives then the binding is sent back immediately.

Grouping nested data structures

The DU algorithm binds only a single record, namely the top level of the structure, and the rest is transferred when the top level binding arrives at a site. The Mozart algorithm binds a complete tree in a single operation. In this way, it avoids the creation of distributed variables for the intermediate nodes. For example, the unification $x_1 = f(g(a))$, is represented in the DU algorithm as three actions $x_1 = f(x_2) \wedge x_2 = g(x_3) \wedge x_3 = a$. In the DU algorithm, the arrival of $x_1 \Leftarrow f(x_2)$, enables the arrival of $x_2 \Leftarrow g(x_3)$, and similarly for x_3. In the Mozart algorithm, the binding $x_1 \Leftarrow f(g(a))$ arrives in one step, so the variables x_2 and x_3 are never created.

Winner optimization

The winner is the proxy that sent a successful binding_request(N). This proxy does not need to be sent N since it already exists on the proxy's site. The proxy can be sent a simple acknowledgement that its binding request was successful.

This optimization avoids the redundant work done by the R-BIND rule (see Section 8.7.2). It requires the following protocol extensions: the extended proxy state INITIATED(N) where N is the binding, the extended message binding_request(N,PId) where PId identifies the winning proxy, and the new message binding_ack from the owner to the winning proxy. When the proxy receives binding_ack, then it retrieves N from the INITIATED(N) state.

Asynchronous streams

A variable that is exported from its owner site can be *preregistered*. That is, the destination site is added to the owner's reglist without waiting for a registration message. This is correct if there is a FIFO connection to the destination site. Preregistering variables allows elements to be added to streams asynchronously. The example of Section 8.3.4 relies on this behavior.

Let us look closely to see what happens. Assume variable `X0` exists on sites 1 and 2. Binding `X0=m1|X1` on site 1 causes `m1|X1` to be sent to site 2. `X1` will be preregistered, i.e., $M_{\texttt{X1}}$.reglist $\leftarrow$ $M_{\texttt{X1}}$.reglist $\cup$ $M_{\texttt{X0}}$.reglist when the binding leaves site 1. If `X1` is bound on site 1, then its binding will be sent immediately to site 2 without waiting for a registration request from site 2.

If preregistration is not done, then adding elements to a stream requires a round trip message delay for each element. This is because remote proxies have to be registered

before they can receive a binding. In our example, binding `X0=m1|X1` on site 1 causes `m1|X1` to be sent to site 2. When it arrives, an `X1` proxy is created on site 2 and promptly registers with site 1. Binding `X1=m2|X2` on site 1 will not send the binding to site 2 until the registration arrives on site 1. Therefore each new element appears on site 2 only after a round trip.

Lazy and eager variables

Lazy and eager logic variables are defined informally in Section 8.3.2. In terms of the on-line DU algorithm, they differ only in the scheduling of the ARRIVE rule. To be precise, laziness is a property of a variable proxy, not of a variable. A proxy is *lazy* if the reduction of ARRIVE is delayed until after INITIATE reduces on that site. If no such delay is enforced then the proxy is *eager*.

In terms of the Mozart algorithm, this is implemented by registering lazy and eager proxies at different times. Eager proxies are registered as soon as they appear on a site (see ARRIVE rule in Figure 8.23). Lazy proxies are only registered after the INITIATE rule is reduced (see Figure 8.22), that is, when a binding request is made.

When two proxies are bound together, the result must be eager if at least one of the two was eager. When a local variable is bound to a proxy, the proxy must become eager if the local variable was eager. Implementing this requires replacing the reg message by three messages: (1) in the ARRIVE rule, reg becomes reg_if_eager, (2) in the INITIATE rule, a new message reg_always is sent, and (3) in the BIND rule, a new message reg_and_make_eager is sent if the local variable is eager.

8.10 Related work

There are two main streams of related work. Some distributed implementations of concurrent logic languages do distributed unification (see Sections 8.10.1 and 8.10.3). Some imperative or dataflow language implementations have a kind of synchronizing variable (see Section 8.10.2). To our knowledge, the present article gives the first formal definition and total correctness proof of a practical algorithm for distributed rational tree unification. The present article also clearly explains for the first time the advantages of using logic variables in a distributed system.

8.10.1 Concurrent logic languages

Many concurrent logic languages have been implemented in distributed settings. These systems do not use logic variables primarily to improve latency tolerance and network transparency. Rather, logic variables are integral parts of their execution models, and the distributed extensions must therefore implement them. We summarize the distributed unification algorithms used in Flat GHC, Parlog, Pandora, DRL, and KLIC.

Flat GHC, Parlog, and D/C-Parlog

Among early implementations doing some form of distributed unification are a Flat GHC (Guarded Horn Clauses) implementation on the Multi-PSI [65], a Parlog implementation on a network of workstations [43], and designs for distributed implementations of Parlog, Pandora, and D/C-Parlog [81, 82]. Pandora extends Parlog with determinacy-driven execution (the Andorra model). D/C-Parlog extends Parlog with linear real-number constraints, namely equations, inequalities, and disequalities. All the above algorithms are defined informally by explaining what happens with arguments of different types. No formal definitions nor correctness arguments are given.

The Parlog implementation contains an algorithm due to [43]. Variables exist on one site and have remote references, which is similar to the owner/proxy model of the Mozart algorithm. Variable-variable unification avoids binding cycles by ordering the variables, as is done in the DU algorithm. All remote references to variables are lazy and dereference chains may cross sites. Preregistering is not done, so asynchronous streams are not possible.

Like early Prolog systems, Foster's algorithm does neither an occur-check nor memoization. When unifying two cyclic structures it may go into an infinite loop. The algorithm has proxy registration (called "ns_read") similar to the Mozart algorithm and a novel form of registration (called "read") that sends the binding only when the variable is bound to a nonvariable term. This is used to get the value for operations that need a nonvariable.

DRL

DRL [35] (Distributed Real-time Logic language) is a concurrent logic language extended with features for distribution and soft real-time control. Distribution is introduced by allowing computations on different sites to communicate through shared logic variables. In DRL, the representative of a logic variable on a site is called a *logic channel*. A logic channel is always statically marked with a direction, which is either output or input. For a given logic variable, only one channel is marked output. Binding the output channel to a term causes the term to appear at all corresponding input channels. The binding blocks until the term contains only ground subterms and logic channels. It follows that variables can be transferred between sites only if they are statically declared as logic channels.

Logic channels can be connected together. This operation is called "unification" in DRL, but the shared logic variables are not actually unified together. To be precise, no variable elimination is done, but communication links are set up between variables. Connecting two output channels causes a future binding of one of them to be sent also to the other. Connecting an input channel to another channel suspends until the input channel receives a value. It follows that dependencies on intermediate sites are not removed.

KLIC

KLIC [44] is an efficient portable implementation of the concurrent logic language KL1 (Kernel Language 1) for distributed and shared-memory machines. KLIC achieves these goals by compiling into C. On one processor running a series of representative benchmarks, the performance of KLIC approaches that of C and C++ implementations. The distributed implementation of KLIC does distributed unification [106], including binding variables to variables. However, the algorithm has several curious properties: binding cycles can be created when binding variables to variables, inconsistencies are ignored, and a variable may be bound to different values on different sites. Apparently, the algorithm is only intended to be used in settings where there is no possibility of inconsistency.

8.10.2 Languages not based on logic

We first compare logic variables with futures and I-structures (see Section 8.10.2), which have been used to improve expressiveness of parallel languages and performance of parallel systems. Then we briefly discuss traditional distributed architectures and how they could be extended to incorporate logic variables (see Section 8.10.2).

Futures and I-structures

The purpose of futures and I-structures is to increase the potential parallelism of a program by removing inessential dependencies between calculations. They allow concurrency between a computation that calculates a value and one that uses the value. This concurrency can be exploited on a parallel machine. To our knowledge, they have not been used in distributed programming. We compare futures and I-structures with logic variables (see also Section 8.3.4).

The call `(future E)` (in Lisp syntax) does two things: it immediately returns a placeholder for the result of *E*, and it initiates a concurrent evaluation of *E* [49]. When the value of *E* is needed, the computation blocks until the value is available. We model this as follows in Oz (where *E* is a one-argument procedure):

```
fun {Future E}
X in
   thread {E X} end
   X
end
```

This can be written more compactly as **fun** {Future E} **thread** {E} **end end**. A future differs from a logic variable in that it can only be bound by the concurrent computation that is created with it. Futures should not be confused with read-only logic variables, which are not tied to any computation. (see Section 8.9.1 and [92]).[6]

An I-structure (for *incomplete structure*) is a single-assignment array whose elements can be accessed before all the elements are computed [12, 133, 64]. It permits

[6] Read-only logic variables are confusingly referred to as "futures" in the cited article.

concurrency between a computation that calculates the array elements and a computation that uses their values. When the value of an element is needed, then the computation blocks until it is available. An I-structure differs from an array of logic variables in that its elements can only be bound by the computation that calculates them.

Two-level addressing

Systems with support for distributed computing commonly provide two-level addressing. This provides the ability to use local and remote references interchangeably. References that arrive on a site are automatically converted to the local form if they refer to local entities. Typical examples include Java RMI [90], CORBA [96], and the Ericsson OTP (Open Telecom Platform) [11, 138].

Two-level addressing can be extended to provide weak logic variables (see also Section 8.3.1). It suffices to add an "unknown" state to variables: (1) threads block when the variable is unknown, (2) when the value is available, all remote references to the variable leave the unknown state, and (3) no forwarding chains are created if a reference travels among many sites. There should be no overhead if the variable is on one site only. To provide full logic variables this is further extended with variable-variable unification. As the CC-Java implementation illustrates, dynamic typing is not necessary (see Section 8.3.5).

8.10.3 Sending a bound term

A basic operation in distributed unification is sending a bound term across the network. [76] investigate the costs of this operation and sketch an algorithm to send only that part of a term required by a consumer. Sending too little increases the message latency, since multiple requests will be done. Sending too much increases network load and memory consumption at the consumer. The proposed algorithm sends exactly that part of a term required by a consumer. For example, a list appending procedure requires only the spine of the list, and not the terms in the list. The algorithm uses "consumption specifications", simple tree grammars extended with an additional terminal `Remote`. These specifications can be given by static analysis or by programmer annotation.

8.11 Conclusions

This article has examined the use of logic variables in distributed computing. We have shown that if the logic variables are well-implemented, then common distributed programming idioms can be written in a network-transparent manner, and they behave efficiently when distributed. We have defined the CU algorithm, a centralized algorithm for rational tree unification, and the DU algorithm, its conservative extension to a distributed setting. The DU algorithm replaces the centralized BIND rule by four rules that do coherent variable elimination. We show that the DU algorithm has good network behavior for common distributed programming idioms. We prove that the DU

algorithm is a correct implementation of unification, and we bound the amount of extra work it can do compared to the CU algorithm. We show that both lazy and eager logic variables are implemented by the DU algorithm. They differ only in the scheduling of a single reduction rule.

We extend both the CU and DU algorithms to the on-line case, in which new equations can be introduced indefinitely during execution. We show that if a weak fairness condition holds and if all variables in the equation satisfy the *finite size* property, then any introduced equation will eventually be entailed by the store.

The Mozart system implements the Distributed Oz language and was publicly released in January 1999 [94]. Mozart contains an optimized version of the on-line DU algorithm. Distributed Oz, also known as Oz 3, conservatively extends Oz 2 to allow an efficient distributed network-transparent implementation [54, 132, 55]. Oz 2 has a robust centralized implementation that was officially released in February 1998 [34]. Oz 3 keeps the same language semantics as Oz 2 and extends it with support for mobile computations, open distribution, component-based programming, and orthogonal failure detection and handling within the language. Oz 2 programs are portable to Oz 3 almost immediately.

8.12 Acknowledgements

This research is funded in Sweden by the Swedish national board for industrial and technical development (NUTEK) and SICS. This research is partially funded in Belgium by the Walloon Region. The development of Mozart at DFKI is supported by the BMBF through Project PERDIO (FKZ ITW 9601).
Author's addresses: S. Haridi and P. Brand, Swedish Institute of Computer Science, S-164 28 Kista, Sweden; email: {seif; perbrand}@sics.se; P. Van Roy, Department of Computing Science and Engineering, Université Catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium; email: pvr@info.ucl.ac.be; M. Mehl and R. Scheidhauer, German Research Center for Artificial Intelligence (DFKI), D-66123 Saarbrücken, Germany; email: {mehl; scheidhr}@dfki.de; G. Smolka, Universität des Saarlandes, D-66123 Saarbrücken, Germany; email: smolka@ps.uni-sb.de.

Michael Mehl implemented the distributed unification algorithm in the Mozart system. Per Brand and Erik Klintskog have extended it to incorporate orthogonal failure detection and handling. Iliès Alouini and Mustapha Hadim gave many valuable comments on this article. We thank the other members of the Mozart projects at SICS, DFKI, and UCL. Finally, we thank the referees for comments that let us vastly improve the presentation.

Chapter 9

A Fault-Tolerant Mobile-State Protocol

A Fault-Tolerant Mobile-State Protocol and Its Language Interface

Per Brand[1],
Peter Van Roy[2],
Raphaël Collet[3],
and Erik Klintskog[4]

[1] perbrand@sics.se, Swedish Institute of Computer Science, S-164 28 Kista, Sweden
[2] pvr@info.ucl.ac.be, Université catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium
[3] raph@info.ucl.ac.be, Université catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium
[4] erik@sics.se, Swedish Institute of Computer Science, S-164 28 Kista, Sweden

9.1 Abstract

Mobile-state protocols are important for distributed object systems. We define a lightweight mobile-state protocol that has a well-defined behavior for site and network failures. The protocol is implemented as part of the Mozart platform for distributed application development based on the Oz 2 language. The protocol provides enough information to the language layer so that we can use the platform to program common fault-tolerant algorithms completely in Oz 2. We formally define the semantics of the network layer and the language interface, and we prove that the protocol correctly implements the language interface.

9.2 Introduction

We define a lightweight mobile-state protocol that has a well-defined behavior for site and network failures. The protocol is part of the Mozart system, a platform for general-purpose distributed application development based on the Oz 2 language [94, 51, 54, 7]. Mozart combines efficient network-transparent distribution with a failure model that allows programming fault-tolerant behavior in Oz 2. The protocol is defined on top of a network layer and provides a well-defined language interface. We formally define both language and network semantics with their failure models, and we prove that the protocol correctly implements the language interface.

Mobile object system. The mobile-state protocol implements a *cell*, an updatable pointer that is the basic stateful entity in the Oz 2 language [132]. Cells are the heart of the Oz 2 concurrent object system [58, 131]. The protocol is a fault-tolerant extension of the distributed mobile-state protocol defined in [132, 131]. In Section 9.6 we prove that the protocol satisfies the cell language semantics if the manager site (see Section 9.5) remains running and accessible and the cell's state pointer is not lost (see Section 9.3.2).

Design goals. The mobile-state protocol has the following design goals. It does not sacrifice performance in the common case of no failures. It has the same performance as the non-fault-tolerant protocol of [132]. When there are failures, it provides enough information to the language layer so that common fault-tolerant algorithms can be implemented there [6]. The failure model is designed to cover the vast majority of failures occurring in general-purpose distributed systems, namely site failures (fail-stop) and both temporary and permanent communication failures [78]. The protocol never makes a unilateral decision about a course of action to take when there is a failure. The language layer, i.e., the application, can always make the decision. For example, the protocol will not time-out by default and it can be interrupted at any time by the application while it is waiting. Time-outs, if desired, are easily installed at the application level. Applications written without fault tolerance in mind may block but will not show incorrect behavior if there is a failure.

Collaborative tool development. One example of the usefulness of this protocol is a high-performance collaborative design tool that we are developing in Oz 2 for geographically-separated design teams [47]. Our current tool consists of 20,000 lines of Oz 2 and runs on the Mozart system. It implements a coherent graphic editor and whiteboard, has high performance even over very slow networks, does full and automatic remote code loading, and is fault-tolerant.

Related work. The language-based approach of this paper has some similarities to the metaobject approach of the FRIENDS system [37] and to the language-based approach of the FT-SR language and system [109]. Two main differences are that mobility is a basic ingredient of our protocol and that we give a detailed formal definition with correctness proof of the implemented protocol.

Outline of the paper. The Mozart system consists of four levels of abstraction (see Figure 9.1): OZL (Oz 2 language layer, Section 9.3), RML (reliable message layer, Section 9.4), DGL (distribution graph layer, Section 9.5), and TCP (standard transport

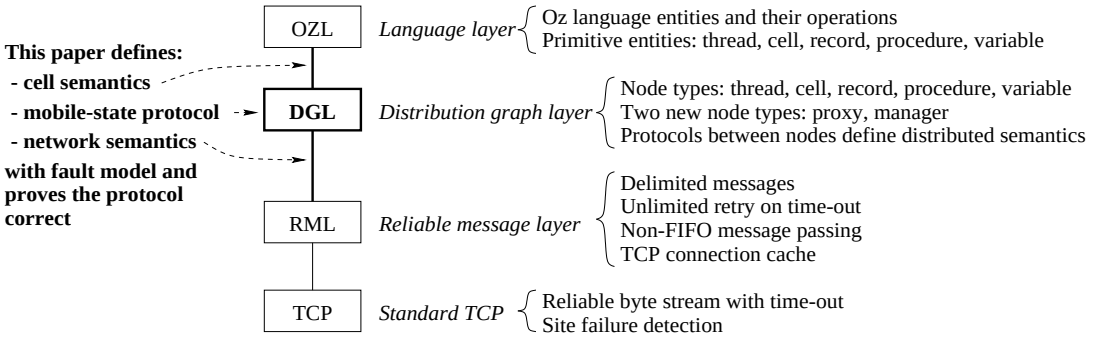


Figure 9.1: Abstractions in the Mozart implementation

layer [27, 139]). The focus of this paper is on the mobile-state protocol, which is part of the DGL. The protocol uses the RML operations to implement cells for the OZL. The protocol guarantees that all sites see the same sequence of state updates. The RML obeys a well-defined failure model. This paper shows how the RML failure model is reflected into the OZL so that fault tolerance can be implemented within Oz 2. All the formal treatments are included as appendices, so that the paper's claims can be verified if desired.

9.3 Language semantics (OZL)

From the application programmer's point of view, the mobile-state protocol implements a *cell*, which is the basic stateful entity in the Oz 2 language [132]. We give a summary of the language semantics, distributed semantics, and fault-tolerant semantics of cells.[5]

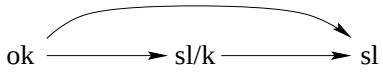
The full language interface of Sections 9.3.3 and 9.3.4 consists of a failure model and three cell operations, `Exchange`, `Probe`, and `Insert`. The failure model is defined in terms of individual cells, where each cell has an internal error state. The operations' semantics are defined with five reduction rules. The rules' operation depends on the internal error state.

9.3.1 Language semantics of cells

The Oz 2 execution model consists of fair, sequential dataflow threads observing a shared store [51]. Cells have two basic operations, which are executed within threads: creating a new cell and updating a cell's content. For brevity, we limit the discussion to the update, called *exchange*. In the call `{Exchange C X Y}`, `C` references a cell and `X` and `Y` are references into the store. The exchange is executed atomically. After

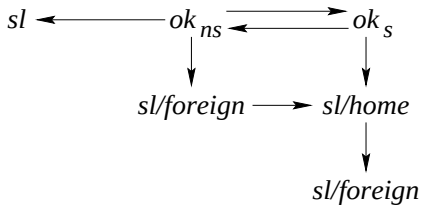
[5] Appendix 9.8.6 gives a full treatment.

Global error states



ok	Normal operation
sl/k	State lost everywhere except site k
sl	State lost everywhere

Local error states (proxy knowledge of global state)



ok s	Normal operation, state at proxy
ok ns	Normal operation, state not at proxy
sl/home	Manager site down, state at proxy
sl/foreign	Manager site down, state not at proxy
sl	State lost everywhere
netdown	Network inactive (add to any of above)

Figure 9.2: Local and global error transitions of a cell

the execution, `Y` is a reference to the cell's new content and `X` becomes a reference to the cell's old content.

9.3.2 Distributed semantics of cells

The distributed semantics is an extension of the language semantics that specifies how the reduction is partitioned among multiple sites. In the case of a cell, there is always one site, say S_j, that contains the current content and that has the right to update the content. We say that this site has the cell's *state pointer*. Executing an exchange on site S_i atomically causes the cell's new content to come to site S_i. This semantics is at the heart of a mobile object system that has the defining property that mobile objects are always executed *locally* [131]. The object state is always brought to the site containing the thread that executes the method.

9.3.3 Cell failure model

The cell failure model is part of the fault-tolerant semantics of cells. Each cell has a global error state and each cell proxy has partial knowledge of this global state (see Figure 9.2). A cell can be in three global states: normal operation (ok), state pointer lost everywhere except on site *k* (sl/k), and state pointer lost everywhere (sl). Locally, if operation is normal, the state toggles between not having and having the state pointer (ok_{ns} and ok_s). If something goes wrong locally, one can know that the state pointer is lost everywhere (*sl*), everywhere except for the local site (*sl/home*), or everywhere except possibly for one other site (*sl/foreign*). In addition to the above local state, one can always know that the network is currently inactive (*netdown*). Generally but

not always, network inactivity is temporary. The proxy cannot determine whether the network will ever become active again.

9.3.4 Fault-tolerant semantics of cells

The distributed semantics of exchange is extended to give a precise behavior when failures occur. Two new operations, `Probe` and `Insert`, are provided to interact with the fault-tolerant side of the protocol. The three operations have the following informal definitions:

- `{Exchange C X Y}` completes atomically if there is no failure. If there is a failure, then the exchange blocks and an optional handler is executed. The handler call may be followed by a retry of the exchange, which is useful in the case of temporary failures.

- `{Insert C H R}` where `C` references a cell, `H` references a one-argument procedure, and `R` references the redo flag (FALSE or TRUE). This installs a handler on the cell `C`.

- `{Probe C F}` where `C` references a cell and `F` identifies a failure when it occurs. The probe blocks until a failure is detected on its site. At that point, it binds `F` to a value that identifies the failure (e.g., `sl`, `ok+netdown`, etc.).

In other words, if there are no failures then exchange is an atomic read-and-write operation. If there is a failure, then probe allows it to be detected asynchronously and insert allows it to be handled synchronously.

9.3.5 Usefulness of Probe and Insert

The `Probe` and `Insert` operations are all that one needs to program standard fault-tolerant algorithms in Oz 2. For example, it is straightforward to install a failure treatment routine that is invoked asynchronously in its own thread when a failure occurs on a particular cell. We call such a routine a *watcher* [54]. Here is an Oz 2 program that installs a watcher:

```
proc {InstallWatcher C Watcher}
F in                    % Declare local variable F
   thread               % Create a new thread
      {Probe C F}       % Wait until failure occurs
      {Watcher C F}     % Invoke watcher
   end
end
```

The watcher is free to use `Insert` to install a handler that can do anything it likes, e.g., replace the operation by an exception if there is a site failure or retry the operation indefinitely if the network is inactive.

Using watchers, we have written a fault-tolerant version of a collaborative graphic editor that is virtually unkillable. The fault-tolerant layer has been added as an orthogonal addition to the application, which was not originally fault-tolerant [6, 47]. The editor runs on multiple sites; one site is special since it serializes all state updates for coherence. If the special site dies, then an election algorithm is invoked among the others [25]. If a non-special site dies, then the locks that it possesses on graphic entities are released. This means that as long as the editor exists on at least one running site, it survives and it can even grow if additional sites are connected.

9.4 Network interface (RML)

This section briefly defines the Mozart network layer.[6] The system consists of a set of nodes communicating through a network. The network is assumed to be asynchronous and unordered (i.e., non-FIFO).[7] We model network inactivity and permanent site failures. Site failures are instantaneous and permanent. Network inactivity is detected instantaneously. The network may or may not recover during the application's execution. Messages in transit from a failed site may be lost. Messages to a failed site are lost. Site failure can cause message loss but network inactivity never does.

There are two network operations. **Send**(N_i:N_j:M) sends message M from node N_i to node N_j. The **Receive**(N_i:N_j:M) is part of a condition-action rule, which is defined as follows:

$$\frac{\textbf{Receive}(N_i{:}N_j{:}M) \wedge \text{Condition}}{\text{Action}} \equiv \begin{array}{l}\textbf{do}\\ \quad \exists\, N_i{:}N_j{:}M \in \text{Net} \wedge \text{Condition} \longrightarrow\\ \quad \text{Net} \leftarrow \text{Net - } \{N_i{:}N_j{:}M\}\\ \quad \text{Action}\\ \textbf{od}\end{array}$$

The network Net is a multiset of messages of the form N_i:N_j:M. In the full formal definition, each site S has associated with it a "local" network Net_S. When the site fails, the messages in the local network are lost. Each rule in the protocol definition is given its own branch in the nondeterministic iteration **do**–**od**. It follows that the protocol is defined according to an interleaving semantics.

The network layer provides two conditions that can be used in rules: **Sitedown**(N) is true iff N's site is down and there are no messages in the network from N. This is a permanent condition. **Netdown**(N) is true iff there is no network activity at N's site. This condition may be temporary. These two conditions are implemented on top of the TCP protocol.

[6] Appendix 9.8.1 gives the full formal definition.

[7] Appendix 9.8.14 shows that FIFO communications are undesirable for network-transparent distributed programming.

9.5 Protocol definition (DGL)

Section 9.5.1 presents the protocol by means of a pedagogically-sound stepwise refinement. Then Section 9.5.2 shows how the three language operations are integrated into the protocol. Figures 9.3 and 9.4 in Appendix 9.8.4 give the proxy and manager state diagrams for the full protocol.

9.5.1 Stepwise construction of the fault-tolerant protocol

We construct the fault-tolerant mobile-state protocol in stepwise fashion from its non-fault-tolerant ancestor by adding new behaviors.[8]

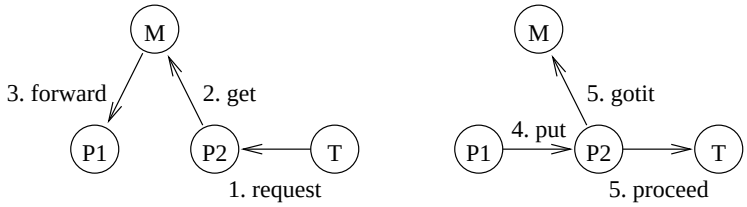
The basic protocol

The basic, non-fault-tolerant mobile-state protocol is the subject of an earlier paper [132]. The protocol is very simple. It manages the movements of the cell's state pointer. Each site has a node called a *proxy*. One special site has a node called the *manager*. When a proxy needs the state pointer to do an exchange, it sends a "get" message to the cell's manager. The manager then sends a "forward" message to the proxy that will eventually get the state pointer. The manager therefore serializes the requests, guaranteeing that there is no starvation. At any moment in time, there exists in the system a sequence of proxies that the state pointer will eventually traverse. This sequence is called the "chain". Formalizing the chain is the key idea used to prove the protocol correct.

Protocol with chain management

The first improvement is to let the manager maintain a conservative approximation of the chain. This is very simple: when the manager receives a "get" message, it appends the requesting proxy to the chain and it sends a "forward" message to the preceding one, so that the latter forwards the state pointer to the requesting proxy.

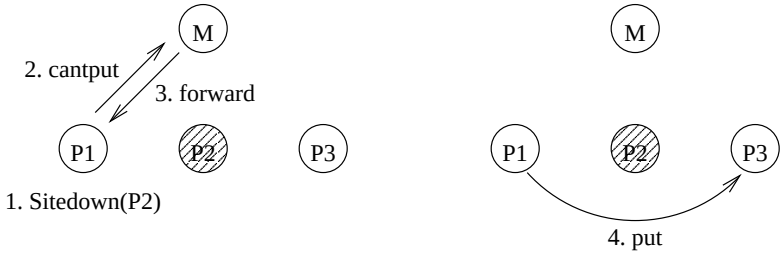
When a proxy receives a "put" message containing the state pointer, it sends a new message, "gotit", to the manager, When the manager receives the "gotit", then it removes from the chain all proxies before the proxy that sent the "gotit".



[8] Appendix 9.8.4 gives the full formal definition.

Bypassing a failed proxy

The second improvement consists of checking whether a proxy is running before forwarding the content-edge to that proxy. Suppose proxy P_1 has to forward the content to proxy P_2. If P_1 detects that P_2 has failed, then it sends a new message, "cantput", to the manager. To which the manager's response is to send another "forward" message to P_1 to bypass the failed proxy. This means that the state pointer is not affected if sites that do not possess it crash.



Determining whether the state is lost

The third improvement is to add an inquiry protocol that permits to detect when the state pointer is definitely lost. This loss can happen in two ways:
1. The state pointer is at proxy P and P's site crashes.
2. The state pointer has been sent over the network in a "put" message and the message is lost because of a site failure (of the sender or the receiver).

Manager failure detection

The fourth and last improvement is to allow proxies to detect a manager failure. This is useful in the following situations:

- Proxy P does not have the state pointer and wishes to execute an exchange. The proxy knows that it will never receive the state pointer and it can directly signal that fact to the thread.
- Proxy P has the state pointer and cannot forward it. The proxy knows that it will keep the state pointer forever.

9.5.2 Definition of language operations

The exchange, insert, and probe operations of the language are defined in terms of message exchange sequences ("protocols") between a thread node and a proxy node. There are three protocols corresponding to the three language operations defined in Section 9.3.4, namely `Exchange`, `Probe`, and `Insert`.[9] We summarize here the most difficult case, `Exchange`. To do an `{Exchange C X Y}`, the thread first sends request(Ny) to the proxy. If there is no failure, the proxy replies with proceed(Nx). If

[9] Appendix 9.8.12 gives full definitions of all three operations.

there is a failure, the proxy replies with skip(H,F) or redo(H,F), where handler H is a one-argument procedure and F describes the failure. The choice between skip and redo is taken when inserting a handler as part of the `Insert` operation. The sequence of messages between thread T and proxy P is a sentence of the following grammar:

Start ::= T:P:request(Ny) *Reply*
Reply ::= P:T:proceed(Nx) | P:T:skip(H,F) | P:T:redo(H,F) *Start*

If the reply is redo(H,F), then the exchange operation is not finished. On the contrary, the thread will send another request to the proxy.

9.6 Correctness

To prove the protocol correct we proceed in three steps. First, we propose a protocol invariant, that is, a succinct characterization of the system state at any moment in time. Appendix 9.8.13 gives a logical statement *I* that extends the invariant of [132] to express all the possible error cases. Second, we need to prove that this statement is actually invariant, that is, that it is initially valid and that the protocol rules preserve its validity (safety theorem). Finally, we need to prove that the protocol does what it is supposed to do, namely correctly move and update the cell's content (liveness theorem). We state the safety and liveness theorems, which are the principal technical results of this paper.

Theorem 9.6.1 (Safety theorem). *The chain invariant I is an invariant of the mobile state protocol.*

PROOF. The proof is not difficult, but cumbersome due to the large number of cases. We verify that the property *I* holds for the initial state and that it is preserved by application of any rule. □

Theorem 9.6.2 (Liveness theorem). *If the cell content is requested at proxy P, then exactly one of the following three statements is eventually true:*
1. *The manager site does not fail and the state pointer is never lost. Then P will eventually receive the cell content exactly once.*
2. *The manager site does not fail and the state pointer is lost before the cell content reaches P. Then P will never receive the cell content, but it will eventually receive notification from the manager that the state pointer is lost.*
3. *The manager site fails before any information reaches P.*

PROOF. We sketch the proof, which divides naturally into three parts. First, if there are no failures in the chain, then we show that the state pointer advances in the chain. This is essentially the proof of [132]. Second, if there are chain failures, then we show that the state pointer correctly bypasses the failed proxies. Third, if the state pointer is lost, we show that this is correctly deduced by the manager. If the manager does not fail, then these three cases are correct. □

9.7 Conclusions

We have precisely defined the fault-tolerant mobile-state protocol in the Mozart platform, which implements the Oz 2 language. The protocol is at the heart of Mozart's concurrent object system. In the case when there are no failures, this protocol has the same network overhead as its non-fault-tolerant ancestor [132]. In the case when there are failures, the protocol provides sufficient hooks to the language layer to allow common fault-tolerant algorithms to be implemented completely within the Oz 2 language [6]. We prove that the protocol correctly updates object state as long as the state itself is not lost and one other site (the manager site) does not crash.

Acknowledgements

This research is partly financed by the Walloon Region of Belgium.

9.8 Appendix

9.8.1 Formal definition of the network layer (RML)

Data

- A set of *site identifiers* S_i, a set of *node identifiers* N_i, and a function site(N_i) giving the site of N_i.
- The *global network* Net, which is a multiset of messages N_i:N_j:M, initially empty. For each site identifier S_k, a *local network* Net_{S_k}, initially empty, which is a multiset of messages N_i:N_j:M where site(N_i)=S_k.
- A set of boolean *site failure* variables sf(S_i) $\in$ {FALSE, TRUE}, initially FALSE. A set of boolean *network inactivity* variables nf(S_i) $\in$ {FALSE, TRUE}, initially FALSE. The latter transit back and forth between FALSE and TRUE. We have nf(S_i)=TRUE iff site S_i detects no network activity during a given time.

Primitive operations

We give the semantics of a reduction rule in terms of guarded commands:

$$\frac{\text{Condition}}{\text{Action}} \equiv \textbf{do}\ \text{Condition} \longrightarrow \text{Action}\ \textbf{od}$$

If there is more than one rule, then each rule is given a branch of the **do**. The primitive operations **PrimSend** and **PrimRec** assume perfectly working Net_{S_i} and Net networks and sites. Failures are modeled at the next level.

$$\textbf{PrimSend}_{S_k}(N_i{:}N_j{:}M) \equiv Net_{S_k} \leftarrow Net_{S_k} + \{N_i{:}N_j{:}M\}$$

$$\frac{\textbf{PrimRec}_{S_k}(N_i{:}N_j{:}M) \wedge \text{Condition}}{\text{Action}} \equiv \frac{\exists\, N_i{:}N_j{:}M \in Net_{S_k} \wedge \text{Condition}}{\begin{array}{l} Net_{S_k} \leftarrow Net_{S_k} - \{N_i{:}N_j{:}M\} \\ \text{Action} \end{array}}$$

PrimSend and **PrimRec** (with Net instead of Net_{S_k}) are defined analogously.

9.8.2 Network layer operations

We define the operations **Send** and **Receive** and the conditions **Sitedown** and **Netdown**. Failures are modeled at this level.

Definition of Send

$$\textbf{Send}(N_i{:}N_j{:}M) \equiv \textbf{PrimSend}_{site(N_i)}(N_i{:}N_j{:}M)$$

with two internal rules:

NORMAL SEND $\dfrac{\textbf{PrimRec}_{S_k}(N_i{:}N_j{:}M) \wedge \neg sf(S_k) \wedge \neg nf(S_k)}{\textbf{PrimSend}(N_i{:}N_j{:}M)}$

LOST SEND $\dfrac{\textbf{PrimRec}_{S_k}(N_i{:}N_j{:}M) \wedge sf(S_k) \wedge \neg nf(S_k)}{\text{skip}}$

The second rule models the fact that when a site crashes, some of the messages sent from this site may be lost in transit. In the real system, such messages are still partly in a site buffer when the site crashes.

Definition of Receive

$$\frac{\textbf{Receive}(N_1{:}N_2{:}Msg) \wedge Condition}{Action}$$

is defined by the following rules:

NORMAL RECEIVE $\dfrac{\textbf{PrimRec}(N_1{:}N_2{:}Msg) \wedge Condition \wedge \neg sf(site(N_2)) \wedge \neg nf(site(N_2))}{Action}$

LOST RECEIVE $\dfrac{\textbf{PrimRec}(N_1{:}N_2{:}Msg) \wedge sf(site(N_2)) \wedge \neg nf(site(N_2))}{\text{skip}}$

Definition of Sitedown

Sitedown(N) is a site activity test that can be used as a condition in a **Receive** rule. It is TRUE iff the site is down and there are no more messages from the site. When executed in node N_i, it has the following semantics:

$$\textbf{Sitedown}(N) \equiv sf(site(N)) \wedge \neg\exists\, N{:}N_i{:}M \in Net$$

This definition assumes that site failure can always be detected.

Definition of Netdown

Netdown(N) is a network activity test that can be used as a condition in a **Receive** rule.

$$\textbf{Netdown}(N) \equiv nf(site(N))$$

If a **Receive** rule contains a **Netdown**(N) condition, then the corresponding "¬nf(site(N))" condition is removed from the semantic definition given above. This allows us to define a protocol that does something instead of nothing when the network is inactive.

9.8.3 Site and network failures

The following two rules are added to the above definitions.

NETWORK ACTIVITY	TRUE
	$nf(S_i) \leftarrow \neg nf(S_i)$
SITE FAILURE	$\neg sf(S_i)$
	$sf(S_i) \leftarrow$ TRUE

The first rule models that a network may become inactive for some time, and then become active again. The network layer implements the inactivity check by sending a "ping" message regularly if no messages are sent or received by the site during a given time. The second rule models a permanent site failure.

9.8.4 Formal definition of the mobile-state protocol (DGL)

We first give the node states (Appendix 9.8.4) and the state diagrams for proxy nodes and manager nodes (Appendix 9.8.4). Then we define the protocol rules. Appendix 9.8.5 defines the basic protocol with chain management. This is the protocol of [132], where the manager is extended to maintain an approximation to the chain. Appendices 9.8.5 and 9.8.5 taken together define an extension of the basic protocol where the state pointer bypasses failed proxies. Appendices 9.8.5, 9.8.5, and 9.8.5 together define a further extension where the manager can determine in some cases that the state is lost. Finally, Appendices 9.8.5 through 9.8.5 define the full protocol, in which proxies also detect manager failure and report it correctly.

Table 9.1 refines the information of Figure 9.2 and defines the function error(P) that is used in the thread interface of Appendix 9.8.5. The table shows what the proxy can know of the global error state. This information is passed to the handler when an error is detected.

Node state

We define the attributes of each node involved in the protocol. The table below gives, for each node type, the name of the attribute, its type and its initial value. T_i are threads, M is manager and P_i are proxies.

Attribute	Type	Initial value
	thread T_i	
id	NodeRef	GetThreadRef(i)
	manager M	
chain	(NodeRef×Identifier)$^+$	(⟨GetProxyRef(1),P_1.requestid⟩)
current	NodeRef×Identifier	⟨GetProxyRef(1),P_1.requestid⟩
knowledge	{OK, INQ, CW, SL}	OK
	proxy P_i	
state	{FREE, CHAIN}	CHAIN(i=1), FREE(i≠1)
content	NULL | NodeRef	N(i=1), NULL(i≠1)
forward	NULL | NodeRef	NULL
backward	NULL | NodeRef	NULL
thread	NULL | NodeRef	NULL
newcontent	NULL | NodeRef	NULL
manager	NodeRef	GetManagerRef()
id	NodeRef	GetProxyRef(i)
requestid	Identifier	newId()
knowledge	{OK, MF, SL, SLMF}	OK

State diagrams

Figures 9.3 and 9.4 give the state diagrams for proxy and manager in the full protocol. These diagrams show three of the refinements as well. The transitions in bold lines give the basic protocol with chain management. Adding the transitions in dotted lines gives the protocol that also determines whether the state is lost. Adding the transitions in thin lines gives the full protocol that handles proxy bypassing and manager failure.

9.8.5 Basic protocol with chain management

P.1. Request content and reply to thread

Receive(T:P:request(T,N_y)) ∧ P.content≠NULL

Send(P:T:proceed(P.content))
P.content ← N_y

P.2. Request content

(a) **Receive**(T:P:request(T,N_y)) ∧ P.state=FREE ∧ P.knowledge=OK

Send(P:P.manager:get(P.id,P.requestid))
P.state ← CHAIN
P.newcontent ← N_y
P.thread ← T

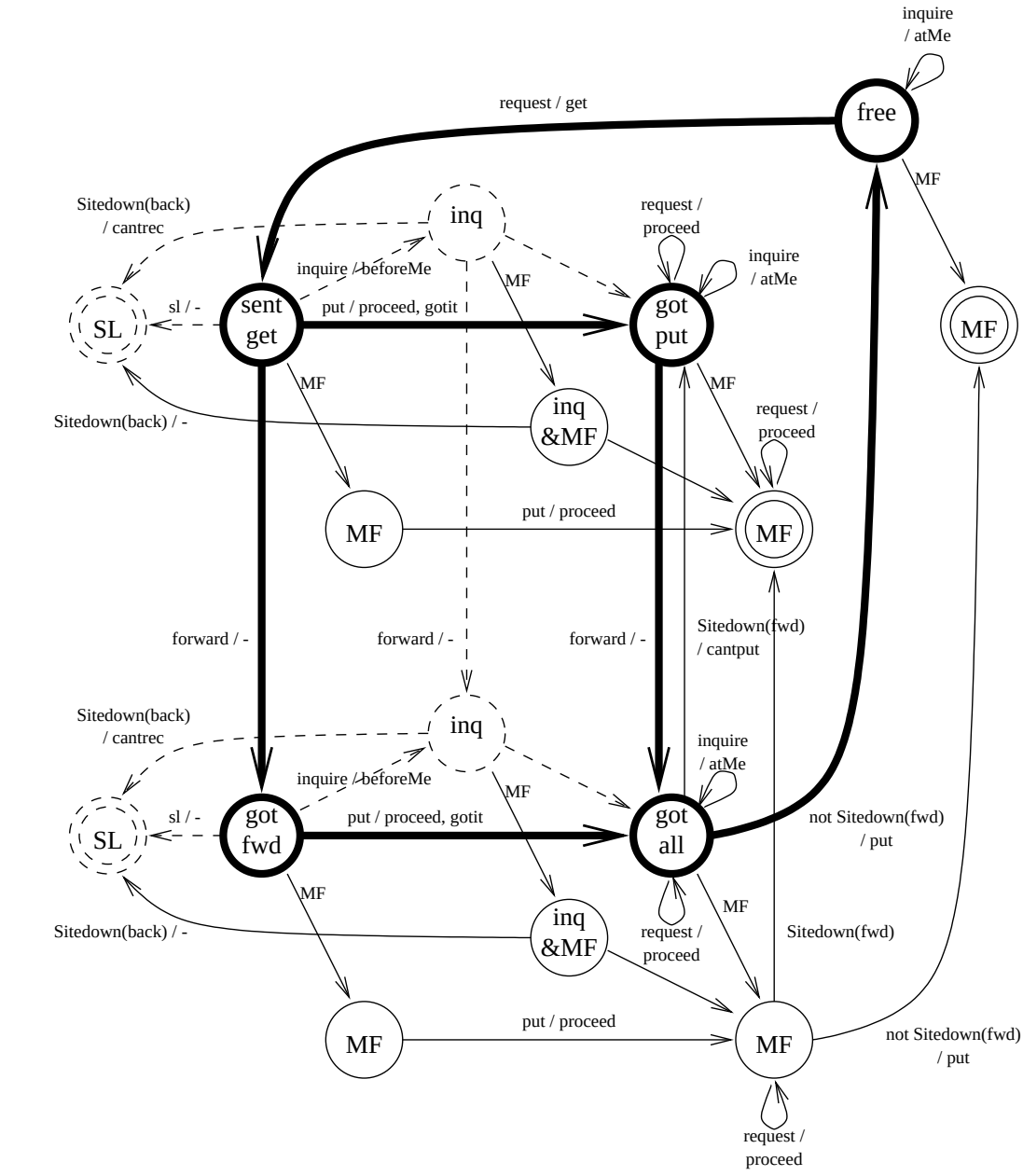


Figure 9.3: Proxy state diagram

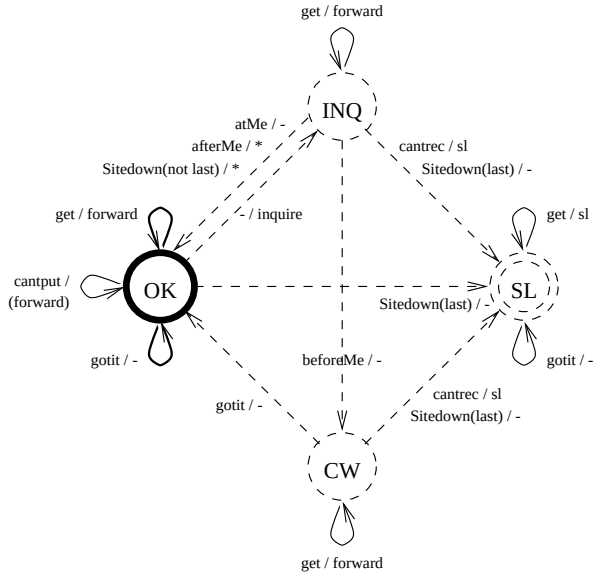


Figure 9.4: Manager state diagram

P.3. Accept content (a) **Receive**(P':P:put(N_Z)) $\wedge$ P.knowledge=OK

Send(P:P.thread:proceed(N_Z))
P.thread $\leftarrow$ NULL
Send(P:P.manager:gotit(P.requestid))
P.content $\leftarrow$ P.newcontent
P.backward $\leftarrow$ NULL

P.4. Accept forward **Receive**(M:P:forward(P')) $\wedge$ P.knowledge=OK

P.forward $\leftarrow$ P'

P.5. Forward content P.content=CVAL $\wedge$ P.forward=P' $\wedge \neg$**Sitedown**(P')

Send(P:P':put(CVAL))
P.state $\leftarrow$ FREE
P.content $\leftarrow$ NULL
P.forward $\leftarrow$ NULL
P.requestid $\leftarrow$ newId()

M.1. Serialize content request (a) **Receive**(P:M:get(P,ri)) $\wedge$ M.knowledge$\neq$SL

Send(M:last(M.chain).1:forward(P))
append(M.chain,$\langle$P,ri$\rangle$)

M.2. Receive gotit

(a) **Receive**(P:M:gotit(ri)) ∧ ⟨P,ri⟩ ∈ M.chain ∧ M.knowledge=OK
erase_before(M.chain,⟨P,ri⟩)
M.current ← ⟨P,ri⟩

(b) **Receive**(P:M:gotit(ri)) ∧ ⟨P,ri⟩ ∉ M.chain ∧ M.knowledge=OK
skip

Bypassing a failed proxy

P.6. Can't forward content

(a) P.content≠NULL ∧ P.forward=P' ∧ **Sitedown**(P') ∧ P.knowledge=OK
Send(P:P.manager:cantput(P.requestid))
P.forward ← NULL

M.3. Receive cantput

(a) **Receive**(P:M:cantput(ri)) ∧ M.knowledge=OK ∧
M.chain=C_1•(⟨P,ri⟩,⟨P',ri'⟩,⟨P",ri"⟩)•C_2
Send(M:P:forward(P"))
erase_element(M.chain,⟨P',ri'⟩)

(b) **Receive**(P:M:cantput(ri)) ∧ M.knowledge=OK ∧
M.chain=C_1•(⟨P,ri⟩,⟨P',ri'⟩)
erase_element(M.chain,⟨P',ri'⟩)

Determining whether the state is lost

P.7. Receive inquire and reply to manager

(a) **Receive**(M:P:inquire(P',ri)) ∧ ri≠P.requestid
Send(P:M:afterMe)

(b) **Receive**(M:P:inquire(P',ri)) ∧ ri=P.requestid ∧ P.content≠NULL
Send(P:M:atMe)

(c) **Receive**(M:P:inquire(P',ri)) ∧ ri=P.requestid ∧ P.content=NULL
Send(P:M:beforeMe)
P.backward ← P'

P.8. Can't receive (detect P.backward failure)

(a) P.backward=P' ∧ **Sitedown**(P') ∧ P.knowledge=OK
Send(P:P.manager:cantrec)
P.knowledge ← SL

P.9. Receive sl

Receive(M:P:sl)
P.knowledge ← SL

M.1. Serialize content request

(b) **Receive**(P:M:get(P,ri)) ∧ M.knowledge=SL
Send(M:P:sl)

M.2. Receive gotit

(c) **Receive**(P:M:gotit(ri)) ∧ ⟨P,ri⟩ ∈ M.chain ∧ M.knowledge=SL
erase_before(M.chain,⟨P,ri⟩)
M.current ← ⟨P,ri⟩

(d) **Receive**(P:M:gotit(ri)) ∧ ⟨P,ri⟩ ∉ M.chain ∧ M.knowledge=SL
skip

(e) **Receive**(P:M:gotit(ri)) ∧ ⟨P,ri⟩=M.current ∧ M.knowledge=CW
erase_before(M.chain,⟨P,ri⟩)
M.knowledge ← OK

M.4. Send inquire

(a) M.knowledge=OK ∧ M.current=⟨P,ri⟩=first(M.chain)
Send(M:P:inquire(NULL,ri)
M.knowledge ← INQ

(b) M.knowledge=OK ∧ M.current=⟨P,ri⟩ ∧ ⟨P',ri'⟩=pred(M.chain,⟨P,ri⟩)
Send(M:P:inquire(P',ri)
M.knowledge ← INQ

M.5. Accept reply of inquire

(a) **Receive**(P:M:afterMe) ∧ M.knowledge=INQ
M.knowledge ← OK
M.current ← succ(M.chain,M.current)

(b) **Receive**(P:M:atMe) ∧ M.knowledge=INQ
M.knowledge ← OK

(c) **Receive**(P:M:beforeMe) ∧ M.knowledge=INQ
M.knowledge ← CW

M.6. Receive cantrec

Receive(P:M:cantrec) ∧ M.knowledge=CW
M.knowledge ← SL
Send(M:P_i:sl) ⋆ *for each P_i after P in M.chain* ⋆

M.7. Detect proxy failure

(a) ⟨P,ri⟩=M.current≠last(M.chain) ∧ **Sitedown**(P) ∧ M.knowledge≠SL
M.knowledge ← OK
M.current ← succ(M.chain,M.current)

(b) ⟨P,ri⟩=M.current=last(M.chain) ∧ **Sitedown**(P) ∧ M.knowledge≠SL
M.knowledge ← SL

Manager failure detection

P.3. Accept content

(b) **Receive**(P':P:put(N_Z)) ∧ P.knowledge=MF
Send(P:P.thread:proceed(N_Z))
P.thread ← NULL
P.content ← P.newcontent
P.backward ← NULL

P.6. Can't forward content

(b) P.content≠NULL ∧ P.forward=P' ∧ **Sitedown**(P') ∧ P.knowledge=MF
P.forward ← NULL

P.8. Can't receive (detect P.backward failure)

(b) P.backward=P' ∧ **Sitedown**(P') ∧ P.knowledge=MF
P.knowledge ← SLMF

P.10. Detect manager failure

Sitedown(P.manager) ∧ P.knowledge=OK
P.knowledge ← MF

Thread interface

P.11. Report error

(a) **Receive**(T:P:request(T,N_y)) ∧ error(P)=F ∧ F≠*ok* ∧ P.hdl=(H,R)
Send(**if** R **then** P:T:redo(H,F) **else** P:T:skip(H,F))

(b) P.thread=T ∧ T≠NULL ∧ error(P)=F ∧ F≠*ok* ∧ P.hdl=(H,R)
Send(**if** R **then** P:T:redo(H,F) **else** P:T:skip(H,F))
P.thread ← NULL

P.12. Probe

Receive(T:P:probe) ∧ error(P)=F ∧ F≠*ok*
Send(P:T:failure(F))

P.13. Insert

Receive(T:P:insert(H,R))
P.hdl ← (H,R)
Send(P:T:insertack)

9.8.6 Formal definition of the language semantics (OZL)

From the application programmer's point of view, the mobile-state protocol implements a *cell*, which is the basic stateful entity in the Oz 2 language [132]. Appendices 9.8.7 and 9.8.8 introduce the Oz 2 execution model and define the cell's operational semantics independent of distribution and fault tolerance. This suffices to reason about programs without mentioning the network. Appendix 9.8.9 refines this semantics by adding a distribution model. This allows to reason also about the distribution behavior in the case when there are no failures. Appendices 9.8.10 and 9.8.11 define a failure model and add it to the semantics. This allows to reason about both distribution and failure behavior.

The full language interface of Appendix 9.8.11 consists of a failure model and three cell operations, `Exchange`, `Probe`, and `Insert`. The failure model is defined in terms of individual cells, where each cell has an internal error state. The operations' semantics are defined with five reduction rules. The rules' operation depends on the internal error state.

Proxy state PS_i (site i) *Description*	Proxy's knowledge of global error state	Value of error(P)
P.know=OK ∧ P.content≠NULL *No known problem, state at P*	ok ∨ sl/i	ok_s
P.know=OK ∧ P.content=NULL *No known problem, state not at P*	ok ∨ sl ∨ (sl/j ∧ j ≠ i)	ok_{ns}
P.know=MF ∧ P.content≠NULL *Manager site down, state at P*	sl/i	*sl/home*
P.know=MF ∧ P.content=NULL *Manager site down, state not at P*	sl ∨ (sl/j ∧ j ≠ i)	*sl/foreign*
P.know=SL *State lost*	sl	*sl*
P.know=SLMF *Manager site down, state lost*	sl	*sl*
Netdown(P) *Network inactive; add to any of above*		error(P)+ *netdown*

Table 9.1: Proxy states and error knowledge (definition of error(P))

9.8.7 Oz 2 execution model

The Oz 2 execution model consists of fair dataflow threads observing a shared store [51]. A *dataflow* thread can execute its next statement only when all values the statement needs are available. Threads contain statement sequences S_i and references into the store. The only way for threads to communicate is through shared references.

The store is a set of two kinds of variables: variables that can be bound only once (called *logic variables*, e.g., X, Y, Z), and variables that can be bound more than once (called *cells*, n). It is important to have both in a distributed system. The distributed protocol for logic variables is much more efficient than that for cells, and logic variables cover most cases of message passing [53]. Dataflow synchronization of threads is implemented with logic variables.

The execution of an Oz 2 program is defined by the reduction of condition-action rules, written as follows:

$$\frac{S_1 \parallel S_2}{\sigma_1 \parallel \sigma_2} \quad C$$

Rule reduction obeys an interleaving semantics, i.e., there is no overlap between rule reductions. A rule becomes reducible when three conditions are satisfied: S_1 is the first statement of some thread, the actual store σ matches the store σ_1 in the rule, and boolean condition C is true. Reducing the rule replaces S_1 by S_2 in the thread. Fairness between threads implies that a rule is guaranteed to reduce eventually when it is reducible and when it refers to the first statement of a thread.

9.8.8 Language semantics of cells

Cells have two basic operations: creating a new cell and updating a cell's content. For brevity, we limit the discussion to the update, called *exchange*. It has the following informal definition:

- `{Exchange C X Y}` where `C` references a cell. The exchange is executed atomically by a thread. After the execution, `Y` is a reference to the cell's new content and `X` becomes a reference to the cell's old content.

Exchange has the following language semantics:

$$\textsc{Exchange}\quad \frac{\texttt{\{Exchange C X Y\}} \;\|\; \texttt{X=Z}}{\texttt{C=n, n:Z, }\sigma \;\|\; \texttt{C=n, n:Y, }\sigma}$$

For brevity, we denote a store of the form $\{\alpha_1\} \cup \sigma$ as α_1,σ.

9.8.9 Distributed semantics of cells

The distributed semantics is an extension of the language semantics that specifies how the reduction is partitioned among multiple sites. We annotate each statement and store content by its site. For example, the statement $\texttt{\{Exchange C X Y\}}_i$ is initiated on site *i*. A cell's value is assumed to exist on exactly one site, e.g., $\texttt{(n:X)}_i$ denotes that cell `n`'s content, namely `X`, exists on site *i*. We call the pair $\texttt{(n:X)}_i$ the cell's *state pointer*, and we assume that the state pointer exists on exactly one site.

For brevity, we omit the site annotation for a binding that exists on each site that needs it. That is, instead of $\texttt{(C=n)}_1$, $\texttt{(C=n)}_2$, ... $\texttt{(C=n)}_k$, we simply write `C=n`, where variable `C` references the cell with name `n`.

With these notations, the distributed semantics of exchange is:

$$\textsc{Exchange}\quad \frac{\texttt{\{Exchange C X Y\}}_i \;\|\; \texttt{(X=Z)}_i}{\texttt{C=n, (n:Z)}_j\texttt{, }\sigma \;\|\; \texttt{C=n, (n:Y)}_i\texttt{, }\sigma}$$

That is, executing exchange on site *i* atomically causes the cell's new value to come to site *i*. This semantics is the heart of a mobile object system that has the defining property that objects are always executed *locally* [131]. The object state is always brought to the site containing the thread that executes the method.

9.8.10 Cell failure model

We define the cell failure model as a prelude to giving the fault-tolerant language semantics of cells. Each cell has a global error state and each cell proxy has partial knowledge of this global state (see Figure 9.2 and Table 9.1). We denote a cell's global state by n:glob(f) and its local state on site *i* by n:loc(f)$_i$. Each cell has one site, the *manager site*, that plays a crucial role in the mobile-state protocol. This is explained in Section 9.5.

A cell can be in three global states: normal operation (ok), state pointer lost everywhere except on site *k* (sl/k), and state pointer lost everywhere (sl). Locally, if operation is normal, the state toggles between not having and having the state pointer (ok_{ns} and ok_s). If something goes wrong locally, one can know that the state pointer is lost everywhere (*sl*), everywhere except for the local site (*sl/home*), or everywhere except possibly for one other site (*sl/foreign*). The transitions from *sl/foreign* to *sl/home* to *sl/foreign* again are due to possible messages in transit (a "put" and a "forward" message, see the protocol definition below). The final *sl/foreign* state is permanent.

In addition to the above local state, one can always know that the network is currently inactive (*netdown*). The complete local state is a pair, e.g., ok_{ns}+*netdown*, *sl/home+netdown*, and so forth. Generally but not always, network inactivity is temporary. The proxy cannot determine whether the network will ever become active again.

9.8.11 Fault-tolerant semantics of cells

The distributed semantics of exchange is extended to give a precise behavior when failures occur. Two new operations, *probe* and *insert*, are provided to interact with the fault-tolerant aspect of the protocol. The three operations have the following informal definitions:

- `{Exchange C X Y}` completes atomically if there is no failure. If there is a failure, then the exchange blocks and an optional handler is executed. The handler call may be followed by a retry of the exchange, which is useful in the case of temporary failures.
- `{Insert C H R}` where `C` references a cell, `H` references a one-argument procedure, and `R` references the redo flag (**`false`** or **`true`**). This installs a handler on the cell `C`.
- `{Probe C F}` where `C` references a cell and `F` identifies a failure when it occurs. The probe blocks until a failure is detected on its site. At that point, it binds `F` to a value that identifies the failure (`sb` or `sl`).

In other words, if there are no failures then exchange is an atomic read-and-write operation. If there is a failure, then insert allows it to be detected synchronously and probe allows it to be detected asynchronously. The exchange can be dynamically replaced by a handler. This allows arbitrary fault-tolerant behavior to be programmed in the language.

The precise semantics of these three operations is defined by five reduction rules. In these rules, the store holds the cell's global error state s_g as n:glob(s_g), the local error states s_l (i.e., per site) as n:loc(s_l)$_i$ for site *i*, and the local handler procedures H and redo flags R as n:hdl(H,R)$_i$. The notation (α_1, α_2) means to reduce α_1 and α_2 sequentially. The notation `{H F}` means to apply the procedure `H` to the argument `F`.

$$\text{EXCHANGE} \quad \frac{\texttt{\{Exchange C X Y\}}_i \;\|\; \texttt{X=Z}_i}{\texttt{C=n, (n:Z)}_j, \sigma \;\|\; \texttt{C=n, (n:Y)}_i, \sigma} \quad \text{n:glob(ok)} \wedge \text{n:loc}(ok\_)_i$$

$$\textsc{Redo}\quad \frac{\texttt{\{Exchange C X Y\}}_i \;\|\; \texttt{(\{H F\};\{Exchange C X Y\})}_i}{\texttt{C=n, n:hdl(H,true)}_i,\sigma \;\|\; \texttt{C=n, (F=}f\texttt{)}_i\texttt{, n:hdl(H,true)}_i,\sigma} \quad \text{n:loc}(f)_i \wedge f \neq ok\_$$

$$\textsc{Skip}\quad \frac{\texttt{\{Exchange C X Y\}}_i \;\|\; \texttt{\{H F\}}_i}{\texttt{C=n, n:hdl(H,false)}_i,\sigma \;\|\; \texttt{C=n, (F=}f\texttt{)}_i\texttt{, n:hdl(H,false)}_i,\sigma} \quad \text{n:loc}(f)_i \wedge f \neq ok\_$$

$$\textsc{Insert}\quad \frac{\texttt{\{Insert C H R\}}_i \;\|\; \textbf{skip}_i}{\texttt{C=n, (R=}r\texttt{)}_i,\sigma \;\|\; \texttt{C=n, (R=}r\texttt{)}_i\texttt{, n:hdl(H,R)}_i\texttt{, }\sigma - \{\texttt{n:hdl(\_,\_)}_i\}} \quad r \in \{\textbf{false},\textbf{true}\}$$

$$\textsc{Probe}\quad \frac{\texttt{\{Probe C F\}}_i \;\|\; \texttt{(F=}f\texttt{)}_i}{\texttt{C=n},\sigma \;\|\; \texttt{C=n},\sigma} \quad \text{n:loc}(f)_i \wedge f \neq ok$$

The EXCHANGE rule updates the local error state to become ok_s.

9.8.12 Formal definition of the language-protocol interface (OZL-DGL)

We define the exchange, insert, and probe operations of the language in terms of message exchanges between a thread node and a proxy node. The proxy definition is given in Appendix 9.8.5. The thread definition is here.

When we say the language executes an operation, we mean that a thread in a running program executes the operation. In terms of the DGL, a thread is a node whose state consists of an instruction sequence and a task stack. The thread node attempts to execute the first instruction in the sequence. It does this by initiating a sequence of message exchanges with other nodes. We call such a sequence a "protocol". An important invariant of the system is that all non-proxy nodes always reference nodes on the same site. Therefore a thread always references a node on the same site, and so we can consider communication between them to be completely reliable.

A cell consists of a set of proxy nodes, at most one per site. Each proxy node has zero or more thread nodes that reference it. A thread executes a language operation on a cell by exchanging messages with a cell proxy. There are three protocols corresponding to the three language operations defined in Section 9.3.4, namely `Exchange`, `Probe`, and `Insert`. In this section we define the protocols that implement these language semantics. Because threads are sequential, it turns out that the protocols can be written compactly as sequential procedures instead of in the more cumbersome reduction-rule notation.

Exchange

The `{Exchange C X Y}` operation is defined by a protocol between its thread and a cell proxy. Infinite sentences are possible in this protocol. This is perfectly acceptable; it means that the application decides infinitely often that it should continue to wait. An application may postpone a decision indefinitely. This means that a complete sentence may not appear during execution. This is perfectly acceptable also. The exchange protocol is defined as follows:

```
function exchange(P,T,Ny)
  Send(T:P:request(Ny))
  return exch_loop(P,T,Ny)
end

function exch_loop(P,T,Ny)
  Receive(P:T:Msg)
  case Msg of
    proceed(Nx) then return Nx
    redo(H,F) then apply(H,F); return exchange(P,T,Ny)
    skip(H,F) then apply(H,F); return NULL
end end
```

Argument P corresponds to the cell `C`, argument T references the thread, and Ny corresponds to `Y`, the cell's new content. If the exchange returns with a non-NULL value Nx, then Nx references the cell's previous content. In that case the thread unifies Nx with `X`.

Probe

The `{Probe C F}` operation is defined as follows:

```
function probe(P,T)
  Send(T:P:probe); Receive(P:T:failure(Nf)); return Nf
end
```

Argument P corresponds to the cell `C` and argument T references the thread. The probe returns with a reference Nf describing the failure. The thread then unifies Nf with `F`.

Insert

The `{Insert C H R}` operation is defined as follows:

```
procedure insert(P,T,H,R)
  Send(T:P:insert(H,R)); Receive(P:T:insertack)
end
```

Argument P corresponds to the cell `C`, argument T references the thread, argument H references the handler, a one-argument procedure, and argument R references the redo flag. The insert operation returns when the handler has been successfully inserted at the proxy.

9.8.13 Protocol invariant

The following invariant is derived from the mobile state protocol of [132]. To simplify the use of the local networks $\mathrm{N}_{\mathrm{S_i}}$, we will use the notation N^*, defined as:

$$\mathrm{N}^* = \mathrm{N} \cup \bigcup_{\text{sites } \mathrm{S_i}} \mathrm{N}_{\mathrm{S_i}}$$

Here is the invariant:

$$
\begin{array}{rcll}
I & \equiv & I_p \wedge I_a \wedge I_b \wedge I_c \wedge I_d \wedge I_o \wedge I_m \wedge I_k & (9.1)\\
I_p & \equiv & \{1,\ldots,k\} = A \uplus B \uplus C \uplus D & (9.2)\\
I_a & \equiv & A = \{a_1,\ldots,a_j\} \wedge j > 0 & (9.3)\\
 & \wedge & \forall i : 1 \le i \le j : \mathrm{P}_{a_i}.\mathrm{state} = \mathrm{CHAIN} & (9.4)\\
 & \wedge & \forall i : 1 \le i < j : \mathcal{N}_{a_i} = \left\{ m \in \mathrm{N}^* \;\middle|\; \begin{array}{l} m = \_{:}a_i{:}\mathrm{forward}(\_) \\ \vee m = a_i{:}\_{:}\mathrm{cantput}(\_) \end{array} \right\} \wedge & \\
 & & \left\{ \begin{array}{ll} & \mathrm{P}_{a_i}.\mathrm{forward} = \mathrm{NULL} \wedge \mathcal{N}_{a_i} = \{\_{:}a_i{:}\mathrm{forward}(a_{i+1})\} \\ \vee & \mathrm{P}_{a_i}.\mathrm{forward} = a_{i+1} \wedge \mathcal{N}_{a_i} = \emptyset \\ \vee & \mathrm{P}_{a_i}.\mathrm{forward} = \mathrm{NULL} \wedge \mathcal{N}_{a_i} = \{a_i{:}\mathrm{M}{:}\mathrm{cantput}(\_)\} \wedge \mathrm{sf}(\mathrm{site}(\mathrm{P}_{a_{i+1}})) \\ \vee & \mathrm{P}_{a_i}.\mathrm{forward} = \mathrm{NULL} \wedge \mathcal{N}_{a_i} = \emptyset \wedge \mathrm{sf}(\mathrm{site}(\mathrm{M})) \\ \vee & \mathrm{sf}(\mathrm{site}(\mathrm{P}_{a_i})) \end{array} \right. & (9.5)\\
 & \wedge & \mathrm{P}_{a_j}.\mathrm{forward} = \mathrm{NULL} & (9.6)\\
 & \wedge & \forall i : 1 < i \le j : \mathrm{P}_{a_i}.\mathrm{backward} \in \{\mathrm{NULL}, a_{i-1}\} & (9.7)\\
I_b & \equiv & \forall i \in B : \mathrm{P}_i.\mathrm{state} = \mathrm{CHAIN} \wedge \mathrm{P}_i.\mathrm{forward} = \mathrm{NULL} \wedge \mathrm{P}_i.\mathrm{backward} = \mathrm{NULL} & (9.8)\\
 & \wedge & \left\{ \begin{array}{ll} & \mathrm{P}_i.\mathrm{knowledge} \in \{\mathrm{OK}, \mathrm{MF}\} \wedge \exists\, i{:}\mathrm{M}{:}\mathrm{get}(\_) \in \mathrm{N}^* \\ \vee & \mathrm{P}_i.\mathrm{knowledge} = \mathrm{OK} \wedge \exists\, \mathrm{M}{:}i{:}\mathrm{sl} \in \mathrm{N}^* \\ \vee & \mathrm{P}_i.\mathrm{knowledge} = \mathrm{SL} \\ \vee & \mathrm{P}_i.\mathrm{knowledge} \in \{\mathrm{OK}, \mathrm{MF}\} \wedge \mathrm{sf}(\mathrm{site}(\mathrm{M})) \\ \vee & \mathrm{sf}(\mathrm{site}(\mathrm{P}_i)) \end{array} \right. & (9.9)\\
I_c & \equiv & \forall i \in C : \mathrm{P}_i.\mathrm{state} = \mathrm{FREE} \wedge \mathrm{P}_i.\mathrm{forward} = \mathrm{NULL} \wedge \mathrm{P}_i.\mathrm{backward} = \mathrm{NULL} & \\
 & & \wedge \neg \mathrm{sf}(\mathrm{site}(\mathrm{P}_i)) & (9.10)\\
I_d & \equiv & \forall i \in D : \mathrm{sf}(\mathrm{site}(\mathrm{P}_i)) & (9.11)\\
I_o & \equiv & \mathrm{StateLost} \Leftrightarrow \neg\exists i : 1 \le i \le k : \mathrm{Owner}(\mathrm{P}_i) & (9.12)\\
 & \wedge & \mathrm{Owner(P)} \Leftrightarrow \left\{ \begin{array}{ll} & \mathrm{P.content} = \mathrm{NULL} \wedge \neg\mathrm{sf}(\mathrm{site(P)}) \wedge \neg\mathrm{sf}(\mathrm{site(P')}) \\ & \wedge \exists\, \mathrm{P'}{:}\mathrm{P}{:}\mathrm{put}(\_) \in \mathrm{N}_{\mathrm{site(P')}} \\ \vee & \mathrm{P.content} = \mathrm{NULL} \wedge \exists\, \_{:}\mathrm{P}{:}\mathrm{put}(\_) \in \mathrm{N} \wedge \neg\mathrm{sf}(\mathrm{site(P)}) \\ \vee & \mathrm{P.content} \neq \mathrm{NULL} \wedge \neg\mathrm{sf}(\mathrm{site(P)}) \end{array} \right. & (9.13)\\
 & \wedge & \forall i : 1 \le i \le k : \left\{ \begin{array}{l} i \neq a_1 \Rightarrow \mathrm{P}_i.\mathrm{content} = \mathrm{NULL} \\ \mathrm{Owner}(\mathrm{P}_i) \Rightarrow i = a_1 \end{array} \right. & (9.14)\\
 & \wedge & \text{There is at most one message put in } \mathrm{N}^*. & (9.15)\\
I_m & \equiv & \exists\ \mathrm{chain\ C} : \mathrm{M.chain} = \mathrm{C} \bullet (\langle a_1, \mathrm{P}_{a_1}.\mathrm{requestid}\rangle, \ldots \langle a_j, \mathrm{P}_{a_j}.\mathrm{requestid}\rangle) & (9.16)\\
 & \wedge & \mathrm{M.chain} = (\langle c_1, r_1\rangle, \ldots, \langle c_m, r_m\rangle) & \\
 & & \Rightarrow \left\{ \begin{array}{l} \exists l : 1 \le l \le m : \mathrm{M.current} = \langle c_l, r_l\rangle \\ \forall i : 1 \le i < l : \mathrm{P}_{c_i}.\mathrm{requestid} \neq r_i \vee \neg\mathrm{Owner}(\mathrm{P}_{c_i}) \end{array} \right. & (9.17)\\
I_k & \equiv & \forall i : 1 \le i \le k : \left\{ \begin{array}{l} \mathrm{P}_i.\mathrm{knowledge} \in \{\mathrm{MF}, \mathrm{SLMF}\} \Rightarrow \mathrm{sf}(\mathrm{site}(\mathrm{M})) \\ \mathrm{P}_i.\mathrm{knowledge} \in \{\mathrm{SL}, \mathrm{SLMF}\} \Rightarrow \mathrm{StateLost} \end{array} \right. & (9.18)\\
 & \wedge & \mathrm{M.knowledge} = \mathrm{SL} \Rightarrow \mathrm{StateLost} & (9.19)\\
 & \wedge & \mathrm{M.current} = \langle \mathrm{P}, \mathrm{r}\rangle \wedge \neg\mathrm{sf}(\mathrm{site(P)}) \wedge \neg\mathrm{sf}(\mathrm{site(M)}) & \\
 & & \Rightarrow \left\{ \begin{array}{l} \mathrm{M.knowledge} = \mathrm{INQ} \Leftrightarrow \left\{ \begin{array}{ll} & \mathrm{M}{:}\mathrm{P}{:}\mathrm{inquire}(\_) \in \mathrm{N}^* \\ \vee & \mathrm{P}{:}\mathrm{M}{:}\mathrm{beforeMe} \in \mathrm{N}^* \\ \vee & \mathrm{P}{:}\mathrm{M}{:}\mathrm{atMe} \in \mathrm{N}^* \\ \vee & \mathrm{P}{:}\mathrm{M}{:}\mathrm{afterMe} \in \mathrm{N}^* \end{array} \right. \\ \exists\ \mathrm{P}{:}\mathrm{M}{:}\mathrm{afterMe} \in \mathrm{N}^* \Rightarrow \mathrm{P.requestid} \neq \mathrm{r} \\ \exists\ \mathrm{P}{:}\mathrm{M}{:}\mathrm{cantrec} \in \mathrm{N}^* \Rightarrow \mathrm{StateLost} \end{array} \right. & (9.20)
\end{array}
$$

9.8.14 Undesirability of FIFO in network-transparent distribution

FIFO connections are inherently undesirable for transparent distribution since they impose dependencies between independent message streams. In particular, a thread sending big messages across a FIFO connection forces other threads to slow down. To show this, let T_1 and T_2 be threads such that T_1 sends big messages of size T and T_2 sends small messages of size t over a network with bandwidth b (bytes/sec). Consider the following two cases:

1. **FIFO**. Assume that T_1 and T_2 send the same number of messages, i.e., each time T_1 sends a message, T_2 sends one message. The rate (number of messages/sec) of T_1 is $b/(T+t)$, which is also the rate of T_2. So the mean rate is $b/(T+t)$.

2. **Non-FIFO**. Assume that both T_1 and T_2 can dispose of a bandwidth of $b/2$. The rate of T_1 is $b/2T$ and the rate of T_2 is $b/2t$. So the mean rate is $(1/T+1/t)b/4$.

One can verify that the mean rate is better with non-FIFO than with FIFO:

$$\frac{b}{4}(\frac{1}{T}+\frac{1}{t}) \geq \frac{b}{T+t}$$

With non-FIFO, each thread is guaranteed $1/n$ of the bandwidth if there are n threads. If a thread does not need this much bandwidth, then its leftover bandwidth is shared among the other threads. With FIFO, a thread has no guaranteed share of the bandwidth. Its share depends on the size of the messages sent by all other threads. A compute-bound thread can be forced to become I/O-bound.

Analogous results hold for latency. Across a well-designed non-FIFO connection, message latency is close to the network's basic latency. Across a FIFO connection, message latency may increase without bounds.

The following analogy may help to clarify things. The south of France is a popular vacation destination, and many northern Europeans go there by car. There are two kinds of roads: the Routes Nationales, with one lane for each direction, and the Autoroutes, with two or three lanes for each direction. Any vacation-goer will tell you that the Routes Nationales are *terrible*. You are forced to go as slow as the slowest caravan. On the other hand, the Autoroutes are *great*. If you meet a slow caravan, no problem–just switch lanes and pass it. A system for transparent distributed computing should behave like an Autoroute and not like a Route Nationale.[10]

[10] Feel free to add a bad pun about the Infobahn.

Part III

Design Philosophy

Overview of Part III

The approach that we take in this part of the thesis is to begin by going back to basics and define what we mean by a programming system and how we judge them. We begin with centralized non-concurrent programming systems and end up with a set of criteria to use for evaluating distributed programming systems. This are then used to evaluate Mozart and other systems.

In chapter 10 we define what is meant by a programming system and establish criteria by which to evaluate such systems. We set the stage by categorizing programming systems into three different types, centralized non-concurrent, centralized concurrent and distributed. The focus of the chapter is on centralized and non-concurrent programming systems, i.e. those programming systems where there is the most community experience. There should be little that is controversial here; rather the chapter is an abstract recapitulation of those criteria that can be used to judge the quality of these programming systems. We find that, on this level of abstraction, that there are three important perspectives by which the quality of programming systems are measured. In centralized non-concurrent programming, most programming systems/languages are by these criteria fairly good - all this is so well-established that this is usually taken for granted. Indeed, we are forced to exemplify programming systems that by our criteria are deficient by historic and artificial examples.

The chapter 11 we discuss concurrent programming systems. We shall see that the same abstract criteria (used in chapter 10) can be used to judge such systems, and, in particular, the extensions that are made to cater for concurrency. There is less community experience here, which is maybe why we find 'poor' systems (according to our criteria) in use today. We shall see that concurrent programming involves units of computation (threads or processes) that interact and synchronize with each other in accordance with what we call the *sharing model*. The three sharing models are message-passing, data-flow, and shared objects (or state). Prototypical examples are Erlang [138], KLIC [44], and Java [120]. Mozart/Oz is unusual in that is designed to cater for all three sharing models.

In chapter 12 we discuss the three sharing models of concurrent programming systems in more detail. Given that Mozart/Oz is unique in catering for all three sharing paradigms we need to address the question of whether all three sharing models are needed. As we shall see, all sharing models are conceptually useful.

In chapter 13 we discuss the necessity of having all sharing models as primitives in the implementation. It can be (and is) argued that they need not all be primitive or native in the implementation. This is because, to a certain degree, one model is

programmable in another. We briefly describe how this is achieved. We note that the programmed abstractions do carry some performance penalties, and do not capture all useful features.

Proponents of mainstream concurrent object-oriented systems, which are single paradigm, often take the view that shared objects are enough. We obviously disagree with this. Some of the arguments for good implementation support for data-flow and message-passing for concurrency (without considering distribution) are beyond the scope of this thesis, having to do with consequences and likelihood of programming errors rather than language expressiveness, or system performance. The practical advantages of declarative concurrency - as opposed to shared state concurrent programming which is prone to deadlocks and race conditions is discussed in [129]. Be that as it may, in this chapter we show that there are some advantages in catering for all three sharing models in a concurrent setting. Perhaps, more importantly, we also lay the groundwork for considering the three sharing models in the context of distribution.

In chapter 14 we introduce distributed programming systems and consider the additional factors that need to be considered for an evaluation based on our three criteria.

In chapter 15 we describe the two known approaches to the design of distributed programming systems. One approach, the *assembler* of distributed programming is to directly reflect up to the user the underlying additional mechanism (other than computation) that forms the basis of distributed programming, message-passing between machines. The other approach is to extend a concurrent programming language/system to distribution, or better yet to integrate the distributed programming model with the concurrent programming model. We call this approach the integrated approach. Integrated distributed programming systems will make use of one or more of the sharing models from concurrent programming. This is only natural as distributed programming subsumes concurrent programming - any distributed programming system also caters for the special case when all threads/processes happen to be running on the same machine.

In chapter 16 the integrated approach to the design of distributed programming systems is evaluated. This may also be seen as an evaluation of the Mozart Programming System for two reasons. Firstly Mozart supports all three sharing models (most others support only one) and thus subsumes many other integrated programming systems. Secondly, Mozart has also gone further than others in smoothly integrating distribution into a sound concurrent framework. In this chapter the criteria established in chapter 10 will be used extensively.

Mozart, it will be argued demonstrates more than the feasibility and usefulness of Mozart itself, one particular distributed programming system, but much more. It demonstrates the viability of the integrated approach in general and lays out the path that distributed programming languages/systems should take to ensure that the system measures up to the established criteria. In this chapter we also show that the in the distributed setting that the case for integral support of all three sharing models is very strong.

In chapter 17 we conclude and briefly consider future work. Here we summarize those aspects of Mozart/Oz that we have shown to be essential in a good distribut-

ed programming language/system. In the final section on future work we first briefly describe features that by all the experience, knowledge and principles expressed in this thesis should have been, but for various reasons never were, incorporated into the Mozart system. Finally, we consider aspects of distributed programming languages/systems that we believe to be open research questions. Here, in the context of integrated distributed programming systems, the path ahead is unclear.

Chapter 10

Programming Systems

In this chapter we will begin with some basic definitions and concepts. Thereafter we will consider the properties and characteristics of programming systems. Our attention will be focused on centralized programming systems, programming systems designed for the programming of a single machine. At this point in time such systems are both better understood and considerably more mature than distributed programming systems. We will attempt to characterize some of the most important criteria for judging the quality of programming systems. These criteria will later be used when distributed programming systems are considered and evaluated.

10.1 Basic concepts and definitions

10.1.1 Distributed and Centralized Systems

A distributed system is a set of autonomous processes connected by a network. We take a general view so the system has the following properties.

- The processes are usually, but need not be, located on different computers.
- The network need not be homogeneous, and communication delays may not only vary greatly between machines, but may be unpredictable.
- The probability of localized faults is not negligible including faults in the network.

Note that we are not limiting ourselves to machines connected by a LAN but include WAN and even wireless networks. The system may be heterogeneously connected where some processes can communicate via a fast local network but others only over slow modems. Also the system extends (or may extend) across administrative domain boundaries.

The contrast to a distributed system is, of course, a centralized system. A centralized system runs exclusively on one machine.

10.1.2 Application Domains

As the whole purpose of a programming system is to provide the programmer with a good tool to build applications we need to consider applications. Throughout this dissertation we exclude hard real-time applications from our consideration. Other applications can be roughly divided into three application domains:

1. Centralized non-concurrent
2. Centralized concurrent
3. Distributed

Centralized applications run on one machine. A centralized concurrent application is an application that is modeled in terms of concurrent processes or threads. From the modeling point of view the processes execute independently of one another, or in parallel, occasionally synchronizing and exchanging information with one another. Usually the processes or threads do not actually execute in parallel, rather the processes time-share the same machine or processor.

Process is the generic term used in theoretical work on concurrency. In concurrent systems the term thread is often used instead. There is some degree of confusion in terminology here. One source of confusion is that the term process is also used for operating system processes. Usually, and we will use the term in this sense, thread is used when the concurrent processes run in the same address space, and process when they do not. We use the term process in the programming system sense, whenever we need to consider operating system processes, we will use the term OS-process.

There are two important differences between threads and processes in concurrent programming systems. The separate address spaces protects processes from each other. But this is only crucial in systems based on unsafe languages like C, where pointer arithmetic can (inadvertently) lead to a situation where one thread modifies another thread's local data. In safe languages like Java, Mozart, or Erlang the issue does not arise. The second difference is important from a performance point of view. As threads share the same address space threads may pass large data structures between one another by reference. Processes will need to create a copy of the data structure in each process address space[1].

Most of the time we will be considering threads/processes that reside within one OS-process. For brevity, when we do not need to make the distinction between processes and threads, where our focus is on units of concurrency we will use the term thread rather than the more cumbersome thread/process.

Concurrent applications rely on some underlying scheduler that in some fashion allots portions of execution time to all runnable threads, i.e. threads that are not blocked waiting for input from other threads. In general there is more than one, so that the scheduler at regular intervals stops the one currently running thread (preemption) and schedules another runnable thread.

[1] This is the case in Erlang even when the Erlang processes reside within the same OS-process - and as Erlang is a safe-language it does not rely on the memory protection of the OS.

Sometimes concurrency is strictly necessary for modeling the application. This is where the interaction between threads is so complex and unpredictable that it would be difficult if not impossible for the programmer to manage this himself. Most simulations are of this category, e.g. agent simulation or discrete-event simulation. Sometimes concurrency is used mainly to achieve fairness. A server, for instance, is continuously receiving requests, and although it is possible to serve only one request at a time, fairness considerations make this undesirable.

Concurrent applications may, of course, actually be executed in parallel. For instance, on a shared-memory multiprocessor concurrent threads may be executed in parallel. It is generally acknowledged that the trend in applications is that concurrency becomes ever more important.

A distributed application is an application that is realized by (or built on top of) a distributed system. Such an application is running on more than one machine. Distributed applications are, of course, inherently concurrent in the sense that there are threads or processes executing concurrently on the different machines. They may or may not be concurrent within a single machine.

The reason that some applications are distributed may be due to an inherent geographical separation of parts of the application, e.g. a chat-room. Alternatively, the application may have been distributed for reasons of scalability, e.g. in a server cluster, where distribution is used to achieve parallelism.

10.2 Characterizing Programming Systems

Programming systems are the tools by which programmers program applications. Programming systems may be categorized by the application domains over which they are suitable. Thus we recognize centralized programming systems that do not provide for concurrency, those that do, as well as distributed programming systems.

The distinction between centralized and distributed programming systems is thus very straightforward, merely reflecting the type of applications that may be developed. However the term programming system is not a precise concept. In particular the boundaries of what should be considered as part of a programming system proper and what lies outside is unclear. We need to clarify this in order to be able to discuss and compare different kinds of distributed programming systems.

10.2.1 Programming Languages, Compilers, and Runtime Systems

Let us consider centralized programming systems. At the heart of a (centralized) programming system one finds a programming language, a compiler, and a runtime system. The programming language is the language in which the programmer instructs the computer. The compiler and the runtime system bridge the gap between the programming language and the machine.

An application developer needs, of course, to thoroughly understand the programming language, but this is not enough. In addition the good programmer needs to have

a mental model of performance. This model of performance includes such aspects as execution speed, memory requirements, and real-time properties. The model may not, and usually does not need to be a strict mathematical model allowing the programmer to predict speed and memory usage exactly, but does need to serve as a guide when choosing between different programming constructs to achieve the same end that differ in performance aspects.

The performance and hence the performance model of a programming system is just as dependent on the compiler and runtime system as they are on the programming language itself. It is quite possible for programming systems based on the same programming language, to differ greatly in performance. This is easily overlooked because mature centralized programming systems, based on the same programming language, rarely actually do differ to any large degree. In the course of time programming systems with poorer performance fall by the wayside, a sort of 'survival of the fittest' for programming systems.

Differences do still exist, of course, and they are not necessarily minor. Later we give some examples. But, we should note, that for the really extreme differences one must look at immature programming systems. The Prolog programming language was invented in the 70s with an implementation based on interpretation on source code level. The introduction of WAM - virtual machine technology in Edinburgh Prolog gave a Prolog system that was faster by several orders of magnitude. All subsequent Prolog systems, e.g. SICStus, were based on this technology and the immature interpreter-based Prolog programming systems disappeared.

It our belief that distributed programming systems of today, unlike centralized programming systems are immature. This will be made clear later. Currently we are leading up to a definition of a programming system and it is important not to exclude immature systems from consideration.

10.2.2 Libraries and Tools

We have said that programming systems consist of a programming language, compiler, and runtime system. But in addition to these the typical program developer will make use of numerous tools and libraries. Should these be considered part of the programming system?

We take the view that tools like debuggers and profilers are not part of the programming system. That we take this view is mainly a matter of definition but can be justified. Tools like these reflect human imperfection in the face of great complexity. Even good programmers write buggy programs and may need a debugger to determine the cause of the problem despite good understanding of the programming language. And even with good understanding of the performance model as well as the programming language writing optimal programs is very difficult and profilers can be of help. The problem is that programs are usually by necessity large and complex. Were programming limited to small *toy* programs there would be little need for debuggers, profilers, analyzers etc.

The situation with libraries is different. Programming systems are generally pre-

sented as having both a programming language and a number of standard libraries. One can ask who decides and by what criteria and given functionality is to be considered a language feature or a library feature. To a certain degree the decisions have been made by the developers of the programming system themselves with criteria based on tradition, pedagogy, importance of the feature, stability of the feature over time, or even upon personal preference or how the programming system was implemented. There may be very good reasons behind the division between language and library features, in the context of learning, using, and maintaining a particular programming system. But this is not necessarily conducive to a comparison of programming systems, particularly when comparing over a wide range of systems and languages.

The one objective feature that can be used to distinguish the different types of libraries is if the library functions are fully expressible in the programming language itself. Libraries that are so are there mainly for convenience. This includes reuse, so this is not to say that this is not important in practice. One example is a library for sorting lists, sets, etc. We take the orthodox view that these libraries are not part of the programming system. We call these kinds of libraries *proper* libraries[2]

Native libraries are libraries that are not fully expressible in the language itself. To include all native libraries, even those dealing with i/o, graphics, and standard operating system services (e.g. file system) in the concept of programming system, makes the programming system very large. Our view, is that it is better to err on the including too much side, rather than too little. This definition may make different programming systems very much alike in aspects where the systems interface with standard operating system services (e.g. file system services). But, when comparing systems, we can quickly skip over the similarities and concentrate on the differences.

To naively exclude all so-called libraries from the programming system proper fails to capture essential properties that are needed when comparing programming systems. For example, the *malloc* library is an essential part of programming systems based on the C programming language, while *new* is considered part of C++, and Java programming languages. In C *free* is considered a library function, *delete* is part of C++, while Java handles memory reclamation automatically and needs no counterpart. Comparison of these programming systems is much simplified if malloc, free, delete and new are all considered C language constructs, i.e. are part of the C programming system. Another example is that while the primitive for locking objects is considered to be part of the Oz programming language, the corresponding function in Java and C# are presented to the programmer as a library function. A final example, is that while the arithmetic operators, plus (+) and minus (-) are usually presented as language features in imperative languages, many declarative programming systems treat them as library functions.

Finally there are libraries or features that in the implementation are native, but could be expressed in the programming language itself. The libraries/features have been made native for performance reasons. As we view the programmer's performance model as an intrinsic part of the programming system we consider these li-

[2]Proper to indicate that this is what you expect from the 'library' intuition, just as is the case for say a U.S. public library- you expect to fi nd books written in English, not French

braries/features as part of the programming language and hence programming system. An example is objects in Mozart. The kernel Oz programming language has no objects, instead objects and classes are programmable abstractions. The semantics of objects is given by the abstraction. Nevertheless objects are native or primitive in the implementation for efficiency reasons. Good Oz/Mozart programmers are aware of this, if only indirectly, as reflected in the programmer's model of performance.

10.2.3 Definition of Programming System

With this introduction we are ready to define a programming system.

A programming system is a system to construct programs within a given application domain that when run on a computer produces the desired behavior. A programming system consists of two parts, the programming language and execution system.

The programming language consists of a set of primitive constructs each with a well-defined behavior (semantics) upon execution. Included in the language are all features/libraries that are native (i.e. not expressible in the language itself).

The execution system is responsible for handling the execution of programs written in the given language and includes the compiler, runtime system, and possibly a virtual machine.

It should be noted that our definition makes the concept of a programming system broad, but objective. For instance, for centralized concurrent applications one programming system would be the JVM together with the threads package offered by the operating system and the Java Enterprise native libraries. Usually the threads package and the native libraries are not considered part of the language. In Mozart, threads and concurrency is built into the virtual machine and is considered part of the language core, but arithmetic is not.

Another example that illustrates the broad scope of our definition of programming language is the security manager of Java 1.2. This is presented to programmers as a library, but the library is not only native but also tightly integrated into the virtual machine. The security manager prevents an untrusted component from subverting a trusted component by traversing the execution stack on potentially dangerous system calls (e.g. opening a file) to ensure that the point of origin is not in the untrusted component.

10.3 Qualities of programming systems

There exist many different programming languages, and they all have their enthusiasts, and there is much debate about the various merits and demerits of different languages. We do not consider this here, but concentrate on some important qualities that are agreed upon by all or most: abstraction, awareness and control.

10.3.1 The quality of abstraction

Whatever the programming system being used, the underlying reality when centralized applications are executed is a processor executing machine instructions. It is possible, but not feasible for anything but the most simple of programs, to program directly in machine code, i.e. code that is directly understood by the hardware. It is just too complicated, with too many low level details to consider, and it is too easy to make an error.

The history of the development of centralized programming systems is the development of ever more layers of abstraction on top of the underlying machine (and on top of lower levels of abstraction). The programmer then develops his/her programs using the abstractions; the abstractions in turn are taken by other programs and transformed directly or indirectly into machine instructions that are run directly on the hardware or interpreted at a low level (emulation).

The main abstraction provided to the programmer is the programming language. The purpose of the programming system is to take the abstract program and to transform it so as to be able to execute the program on hardware. We need to give a name to this quality of a programming system, and we will use the term abstraction. Everything else being equal the higher level of abstraction the better the programming system.

Abstraction may serve to provide the programmer with constructs that are simpler and more powerful than what the corresponding machine instructions would be. Sometimes the abstraction goes further than this and low-level details are completely abstracted out, i.e hidden. Typical example of hiding are details of memory layout, register use, and sometimes memory management.

Automatic memory management illustrates the quality of abstraction at its highest level. Deallocation of structured memory is done automatically by the system in the process of garbage-collection, relieving the programmer of all responsibility for memory management. All other things being equal a programming system with garbage-collection is by virtue of greater abstraction superior to one without, i.e. Java is superior to C++ or C.

Because all other things are not equal, C (or C++) is not a dead programming language. For a full comparison, we need to consider the two other important qualities of good programming systems.

10.3.2 The quality of awareness

The programmer also needs a model of performance of the program constructs. The programmer needs this model to be able to choose between different programming techniques to achieve the same goal. The model of performance in a centralized programming system must include both aspects of execution speed and memory use. This model need not be and rarely is a precise mathematical model letting the programmer predict the performance of his/her program exactly. The performance model does, however, have to provide the programmer with a rough idea of performance, at least for programming in the small. Large complex programs are more difficult and additional

support in the form of tools may be required.

We will use the term awareness for this quality. Good awareness means that the performance of the program is reasonably predictable and understandable. Without awareness, without a good performance model, the programming system would be a black-box that the poor programmer could do little more with than insert his/her programs and hope for the best.

For example most C++ or Java programmers know that iteration is more efficient than recursion. There is a major difference in space complexity for most C++-programming systems [3]. Also, while the time-complexity is the same, recursion is slower due to the greater overhead of subroutine calls by some factor. The programmer will therefore tend to use iteration rather than recursion whenever possible, at least for time-critical portions of his/her program.

In this case the poor performance of recursion over iteration is a property of the programming system rather than the programming language per se. Few C++-compilers support tail-call optimization. The Java Virtual Machine does not support tail-call optimization either. If they did so by the best state-of-the-art methods there would not be any difference between tail recursion and iteration. We note that tail-call optimization is supported in C#, to the extent that there is no difference in space-complexity.

10.3.3 The quality of control

The abstraction and hiding that the programming system provides is not, in general, without some cost in terms of performance. Theoretically at a lower level you can achieve all that can be achieved at a higher level, and almost always more. In practice there are all kinds of reasons that some performance is lost in the various translations from higher-level constructions to low-level ones. One reason is that the law of diminishing returns sets in for compiler writers, another that various program invariants that a compiler might use for optimization are not even expressible in the programming language (e.g. that two C variables cannot be aliased).

We call this performance difference between high and low level, the performance gap. For centralized programming systems the reference low level is assembler. At this point this may seem academic, programming in assembler is generally impractical, and we usually want to compare programming systems with each other. Our reasons for this comparison will become clear later. For now we make two observations. First, If we compare the performance of two different programming systems we are in a sense measuring their relative performance gap. Second, if we can show that some programming system has a small or negligible performance gap then from the performance point-of-view we can be satisfied even without considering other existing or even potential programming systems.

The contrary is, of course, also true. If we can show that some programming system has a very large performance gap then we can conjecture that the programming system is either doomed to die or to evolve. Maybe better transformation techniques, i.e.

[3] but Gnu C++ does last-call optimization

better compilers or better runtime systems, will improve the performance to acceptable levels. Alternatively, maybe the abstraction, the programming language, needs to be augmented. Furthermore, a low performance gap would seem to be a necessary trait if a programming system is to become popular (though by no means sufficient).

We will use the term control to describe this quality of a programming system. A system with good control has low performance gap, and a system with poor control has a large performance gap. The intuition behind this term is that the programmer should have sufficiently good control to write programs that perform reasonably well.

An example, of a programming system with poor control was the first Prolog (logic-programming) system of the early 70s. Execution proceeded by source-level interpretation and the language was of little interest until the development of the Warren Abstract Machine (WAM), a virtual machine for Prolog, around 1980. Suddenly the performance gap for transformational programs decreased from a factor of several hundreds to approximately three (later improved by native-code compilation to one). The key factor here was the development of the proper implementation techniques.

Another example concerns the lack of state (mutables). Declarative programming languages, i.e. both logic programming languages like Prolog, and functional programming languages like ML had for a long time no provision for true state. Stateful change could be expressed in these systems but it was not true state in the sense that the implementation did not actually change the value of memory cells. For many programmers with an imperative programming language background, where state and destructive assignment was used everywhere, these programming languages seemed very strange. In particular, some of the techniques to work around the lack of true state (e.g. difference lists) were tricky, and while they served their purpose, they were motivated solely by efficiency considerations in a declarative programming environment. The declarative programming language community did, however, show that state was not as necessary as might be at first supposed over a significant range of applications.

Nevertheless, the lack of mutable state was a severe handicap for many applications. The performance gap was on the level of algorithmic complexity. For example, in stateful systems array updates can be achieved in constant time with no space overhead, whereas in early Prolog systems the time and space complexity was linear in array size[4]. During the course of time many of these types of programming languages have been augmented with provisions for true state. These augmentations involved major changes in the model that programmers used, so in a sense these systems represent different programming languages than the pure declarative programming base upon which they were first built.

The reason that we are belaboring centralized programming systems with poor control is that, when we later extend the notion of control to distributed programming systems, we do find that many systems of today offer very poor control. For the most part we take good control in centralized programming systems for granted, which is fair enough given 50 years of community experience, but it may blind us to the fact that we cannot take this for granted when it comes to distributed programming systems.

[4]In favorable cases this could be improved to logarithmic by use of tree data structures - but there was no way - within the language to keep the tree balanced, so there were no guarantees

We give one final, but artificial, example of a centralized programming system with really poor control. At first sight the system is absurd, but the question we should ask, is if there exist distributed programming system counterparts, if it can be so that popular distributed programming system are as deficient in control as the example.

Imagine a variant of C++, call it C–, that did not provide iteration, but only recursion. From a language point-of-view, considering only expressivity this might be considered legitimate; recursion is more general and iteration can always be expressed as recursion, but not necessarily vice versa (without tedious explicit stack operations). One could even argue that the language is simpler than C++.

If our fictitious C–programming system did not provide good tail-recursion optimization the system would be very deficient in the control aspect. The space requirements for a deep stack would be unacceptable, and the programmer would be tempted (or required) to program on a lower level, i.e. include his/her own assembler routines to achieve iteration. If the C– programming system did provide tail-recursion optimization things are little different; the lack of control is not on the level of algorithmic complexity, but is once again a constant factor.

10.3.4 How good control is needed?

An important question is how good control is needed, or put another way, what is an acceptable level of performance gap.

At first sight is seems clear that a performance gap on the level of algorithmic complexity is rarely acceptable. When we look at general-purpose centralized programming systems used today we find no differences on this level when it comes to memory or execution overhead. The only examples that we can find on such programming systems are historical, these systems being superseded by better systems. We saw earlier that programming systems must either cater for tail-call optimization for recursion or provide for iteration. Declarative programming languages usually have some provision for dealing with state.

We should note however that a performance difference does exist between programming systems with automatic memory management and those without when it comes to real-time properties. Even with the best garbage-collection techniques there is a difference. For certain types of hard real-time applications this is crucial and they are (and should be) programmed in languages like C.

But what about a constant factor performance gap? This is even less clear-cut and no precise answer is possible. In the end, the question of acceptable performance gap is application-dependent. Nevertheless, we will try to give a ball park estimate.

In the mid 80s the author during undergraduate studies worked as a programmer at a company that will remain nameless. The task was to hand-code various time-critical portions of a large program (or more accurately to edit the compiler generated assembler). This was done in a desperate attempt to improve the performance. Presumably this was done at some expense, as the author was one of many involved, yet was done with the awareness that the best that could be hoped for was at best 20% improvement. Such activity would seem to be very uncommon today. Of course, since then not on-

ly have compilers improved, but hand-coding has become more difficult with modern microprocessors.

We see a general trend where the emphasis is more on reliability and/or ease of program development rather than on optimality. This is undoubtedly due to three factors, improved hardware, improved tools (e.g. better compiler and better garbage-collection techniques) and the greater complexity of today's applications.

A good example is the rise of Java in the 90s. For the first time a language that hides memory management, i.e. has garbage-collection, entered the mainstream (though they had been around in academia for quite some time). Explicit memory management on the application level, with C, C++, and many other languages, made for many programming errors. Hiding memory management, with implicit allocation and garbage collection, both eases program development and makes for fewer errors. This has a performance penalty in execution time but has become acceptable for a wide range of applications.

The best state-of-the-art emulation techniques (i.e. byte code interpretation) usually carry a performance penalty of roughly 3 times. If this were generally acceptable, there would not have been the interest in JIT technologies that there are.

For centralized programming we can therefore say, and this is a really rough estimate, that a performance gap between 1.2 to 3 is often a fair trade-off. This does not mean that larger performance gaps would always make the system uninteresting, we might be willing to pay a larger price for really useful abstractions for some types of applications.

10.3.5 The challenge in developing programming systems

Low-level programming languages/systems invariably provide both good awareness and control. More generally the lower the level of abstraction the easier it is to ensure awareness and control. At the extreme lower end, good awareness and control are automatic when the level of abstraction is zero, i.e. programming in assembler.

The challenge when developing programming systems is thus twofold. Firstly, the abstractions offered must be expressive and useful. This would seem to be main point judging by the multitude of books expounding the virtues of various programming systems/languages from this point of view. However, there is also the second challenge; that the abstractions offered are obtained without sacrificing awareness and control. When it comes to centralized programming systems for we have come to expect, and rightly so with 50 years of experience, that all have reasonably good awareness and control at least when it comes to non-concurrent applications.

10.4 Concurrent programming systems

Concurrent programming systems allow the programmer to model a number of concurrent or parallel activities (threads). Concurrency is thus a form of parallelism on the level of the program structure. The threads executing in parallel may, from time

to time, exchange information and synchronize but for much of the time they execute independently.

Concurrency is supported by virtually all operating systems. In the early days of computing when most machines were multi-user machines such support was, of course, necessary. (As it is these days, when the single-user wants to do many things at the same time). Concurrency is also useful on the application level. Any simulation of real world activities will be concurrent, as the real world is highly concurrent.

The poor man's concurrent programming system is to use the concurrency offered by the operating system together with inter-process communication. We will not consider this in detail here, other than to say that this is generally considered unacceptably heavyweight, inefficient and cumbersome as a general purpose concurrent programming system. A large variety of concurrent programming systems/languages have been developed to alleviate this problem.

In some sense, the goals involved in achieving concurrency and parallelism are diametrically opposed. Concurrent systems strive to provide the illusion of many processing units or machines on a single machine. Parallelism, on the other hand, is often concerned with splitting, possibly automatically, a single-threaded application into concurrent units as a preparation for running them in parallel on multiple processors.

Concurrent centralized programming subsumes traditional centralized programming. Traditional (non-concurrent) programming is, from this point-of-view, concurrent programming with a single thread or process.

In the next chapter we will consider concurrent programming systems in more detail. We will evaluate them according to our three criteria and describe the three different sharing models upon which they are based.

10.5 Distributed programming systems

A distributed programming system is a programming system specifically designed and tailored for developing distributed applications. Distributed programming systems do, of course, subsume centralized programming systems. Local computation remains the same. The task of programming the behavior of one site is, at least potentially, relatively unchanged except where and when the site is engaged in coordinated activity with other sites.

Does distributed programming subsume concurrent (centralized) programming? In one sense this is fairly obvious, as concurrency is conceptual parallelism, even if implemented via time-sharing on a single processor machine. Distributed applications, will usually also be concurrent (and parallel), with activity on various sites proceeding concurrently.

Clearly distributed programming systems subsume OS-process based concurrent systems. In this case the operating system acts as the (pre-emptive) scheduler. If the OS-processes in a distributed application all reside on the same machine, we are back to our poor man's concurrent programming system.

However, in another sense, distributed programming systems need not subsume

concurrent programming systems. Distributed programming systems do not necessarily need to support concurrency within a single OS-process. In some distributed applications there is no need for this, the only concurrency that exists is between (processes on different) machines.

Although distributed programming systems need not subsume concurrent programming systems, they often do. For example, in almost all client/server applications there is considerable concurrency on the server side. When the server is a cluster for scalability reasons there is still concurrency within each machine. Telecom switching software, e-business, Internet messaging and gaming, Web services, to name but a few, are all distributed applications that need both types of concurrency.

A distributed programming system that does not cater for concurrency within a machine or OS-process is deficient in both abstraction and control. The programmer does not have the abstractions to achieve what he/she wants to do on a single machine though it is possible. Furthermore performance (control) is poor when the only concurrency available is on the level of the operating system. When his application calls for fine-grained concurrency the application developer will be tempted to program his/her own scheduler.

We will therefore focus on distributed programming systems that have a concurrent core, i.e. that provide for concurrency within an OS-process as well as between machines. We defer further discussion of distributed programming systems to chapters 14 and 15.

Chapter 11

Concurrent programming systems

In this chapter we discuss and evaluate concurrent programming systems according to our three criteria of abstraction, awareness and control. The focus here is on the concepts and constructs that distinguish concurrent programming from centralized sequential programming. Concurrent programming subsumes sequential programming so the discussion and evaluation the we made in the previous chapter still holds - but now there are additional factors to consider.

11.1 Abstraction

Concurrent programming systems provide the programmer, to some degree, with the illusion of many processing units. In the thread model, these processing units share the same memory. In the process model they do not. When considering the quality of abstraction there are two considerations. First, how well the illusion is supported, and second, the quality of the abstractions that govern thread interaction. Discussion of the latter, thread interaction, is deferred until the next section.

On a single processor, only one thread or process can actually be executing at any one time. The key to realizing concurrency is to share the processing unit between the threads. So, at any one time, there exists in the system (at most) one running thread, and multiple runnable and suspended threads. Runnable threads are threads that are waiting to be executed, while suspended threads (if there are any) are those that are currently blocked, waiting for some external or internal event. The illusion of concurrency is achieved by stopping the currently executing thread and scheduling one of the runnable threads.

The software component that chooses which of the runnable threads is to be executed next is the scheduler. One option is to require the programmer to program his/her own scheduler. A simple and fair strategy is to schedule threads in a round-robin fashion. Other strategies that prioritize between threads may also be needed.

Clearly, by the principle of abstraction it is better that the scheduler be built into the programming system. This relieves the programmer of the burden of programming and debugging the scheduler. The round robin scheduling strategy can be made the

default, augmented by a model for assigning thread priorities. Today most concurrent programming systems have a built-in scheduler.

We also need to consider how the currently executing thread is stopped to enable the scheduling of some other runnable thread. One model is to require that the thread explicitly yields control to the scheduler (e.g. the programming system consisting of Rational Rose RT/C++). One drawback, of course, is that if a thread never yields (by programming error) or yields very late, this prevents other threads from making progress.

The alternative is that the system can stop the running thread at regular intervals. This is preemption. There is a trade-off between the overhead of preemption and the advantages of frequent task-switching. In the overhead of preemption is the cost for stopping one thread, saving the state of the thread, choosing another thread, and performing the appropriate initialization from the saved state of the new thread. Clearly preemption is preferred, as the programmer no longer has to annotate his/her program with appropriate yields and fairness is assured. This is much like the advantages of automatic memory reclamation, where the programmer doesn't have to think of freeing memory and is assured that there will be no memory leaks.

Another consideration is if threads or processes can be created dynamically or on the fly or whether this is done statically at compile time. Clearly, a system that permits dynamic creation of threads is strictly more powerful than one that does not.

The centralized precursor of Mozart, the concurrent programming system Oz, was specifically designed for concurrency, and has all the advantages of the features described in this subsection.

11.2 Awareness and Control

11.2.1 Processes versus Threads

Clearly it is important to be aware of the difference between the thread and process models of concurrency. For unsafe languages (with pointer arithmetic) the process model provides memory protection, threads do not. For safe languages OS-support for memory protection is unnecessary and it is both possible and desirable to provide for process support on the language level within one OS-process.

However there are large differences in the performance model between the two paradigms. As representatives, consider two languages that were specifically designed to support concurrency, Oz (or Mozart) supporting threads, and Erlang supporting processes.

With threads, data structures can be quickly be (by reference) passed from one thread to another. With processes, this is much slower, as the data needs to be copied. On the other hand, garbage collection (memory reclamation) needs to be performed for all threads within a process collectively. The most straightforward method is to stop all activity and garbage-collect the entire memory. However, for memory-intensive applications, this takes time, so the system has poor real-time properties. Sophisticated

(and complicated) garbage-collection schemes (e.g. generational, incremental, train algorithm) can help to some degree, but there is still a difference.

From the control perspective concurrent programming systems should provide for both concurrent threads and processes. However, we do not know of any such systems.

11.2.2 Lightweight versus Heavyweight Threads

The performance model of concurrent systems needs to provide a rough model of the cost of threads, both in terms of processing power and memory. This is part of awareness, and most systems seem to have this (even if the performance model comes from the user community and not from the developers of the programming system).

The systems of today, however, vary considerably in the control aspect. This has to do with the cost of threads, the cost of the thread abstraction. The two most important costs are task-switching and memory requirements, which are unnecessarily high in most systems. In particular this applies to mainstream systems such as Java (e.g. JEB) and .NET.

That the costs of threads are unnecessarily high is proven by Mozart/Oz. We say that Mozart threads are lightweight, as opposed to the heavyweight threads found in other systems. In mainstream systems, on a PC, the maximum number of threads that can be created without breaking the system or running out of memory is less than a thousand, in Mozart/Oz the limit is on the order of a million [94]. Also, task-switching is fast enough that the difference in execution time between performing two large tasks concurrently and sequentially is negligible.

As discussed earlier, what do we expect when good abstractions like threads but deficient in control are offered to programmers? Programmers will tend not to use them, rather they will move down one level of abstraction and work there. This is exactly what happens. Rather than making use of thread abstraction, wherever and whenever concurrency needs to be modeled, the programmer does, to a certain degree, his/her own scheduling and control-yielding. A pool of threads is created at start, but not too many as each thread requires a considerable amount of memory. Threads are taken from the pool, upon need, but if more threads are required than are in the pool, tasks are queued. Long-lived tasks cannot be assigned to a thread and let go, but must be programmed to yield control to the thread pool at regular intervals to achieve fairness.

Of course, there is still some true concurrency as there is more than one thread in the thread pool. The programming model is, thus, a hybrid one, with some concurrency provided by the system and some managed explicitly. This is analogous to programming procedures partly in a high-level language and partly in assembler (which is much rarer today, than say, 20 years ago).

From the control aspect it is also important that the difference in execution time between performing two large tasks concurrently or sequentially is small, as otherwise the programmer may be tempted to transform that which is naturally modeled concurrently into a sequential computation, i.e. do his/her own concurrency management.

11.2.3 Conclusion

We saw earlier that the various non-concurrent programming languages/systems show a reasonably high level of maturity by our criteria. They all demonstrate the qualities of abstraction, awareness and control. There are still clear differences between systems/languages in abstraction and control but these differences represent fundamental tradeoffs. The prototypical example is the difference between languages/systems with automatic memory reclamation and those without. One group offers higher levels of abstraction, the other group offers better control, and there is no known way to 'get the best of all possible worlds'.

In this section we saw that concurrent programming languages/systems are not quite in the same state of maturity. There are still systems in use where we can see clear unnecessary deficiencies in the qualities of abstraction and/or control (i.e. that do not reflect fundamental tradeoffs). This, reflects, we believe, the fact that there is less community experience with fine-grained concurrency. Indeed, an awareness of the need for fine-grained concurrent programming systems is fairly recent in the IT-sphere and is connected with the rise of the Internet or distribution (e.g. servlet engines). (We can note that Erlang which was developed in the early 1990's originated in the telecom world where the need for fine-grained concurrency goes back much further).

Chapter 12

Three Sharing Models

The threads (or processes) of concurrent programming systems need to be able to interact with each other. Without the possibility of such interaction the threads could only run in independently in parallel and never influence the action of other threads. Sometimes this interaction is tightly synchronized in the sense that one thread is passive (or blocked) until some other thread performs the appropriate action. Interaction does not necessarily need to be synchronized - the action of one thread can influence the choice (i.e. branch) that another thread takes at some later time.

To our knowledge there exist three and only three distinct different mechanisms for thread interaction in concurrent programming systems. In this chapter we will briefly describe them. The perspective that we take is to consider this from the point of the view of the nature of the language entities that the threads share.

12.1 Sharing models

We use the term *language entity* as a generic term to denote those things that may be manipulated, used and referenced within a programming language. Primitive types (e.g. integers), objects, records and functions are all examples of language entities in various programming languages/systems.

Our focus here is on high-level programming languages, i.e. referentially safe languages with automatic memory management. Some language entities are referenced exclusively by a single thread, while other entities may be shared. Threads interact with each other by acting upon the shared entities.

We can categorize concurrent programming mechanism into three radically different types by the *sharing model*.

1. Object-oriented

2. Data-flow

3. Message-oriented

Reference systems that correspond to the three types are Java- based (object-oriented), the subset of Oz/Mozart without objects and ports (data-flow) and Erlang (message-oriented). We choose these three as they have all been extended to distribution as we shall see in later chapters. For now we limit ourselves to the centralized (but concurrent) scenario.

In object-oriented sharing, threads share an object. The object encapsulates state. A thread may act upon the object (e.g. call a method) changing the value of the state. This will affect other threads in their invocations. Often, objects are protected by locks to ensure that at most one thread is active within the object at any one time.

In message-oriented sharing threads share *mailboxes*. Threads interact by sending messages to one another. A message is stateless, once sent it cannot be changed or retracted. Threads regularly check their mailboxes.

In data-flow sharing threads share a single-assignment or data-flow variable. Threads may be blocked waiting for the data-flow variable to be bound. When one of the threads that share the variable binds the variable (and this can, of course, only be done once) the value is made visible to other threads[1].

In centralized concurrent programming systems threads, upon creation, are typically given an initial set of references, some of which are shared with other threads. Note that the described sharing mechanisms provide for all the dynamicity in sharing. A thread can only be given a reference to a language entity that it currently does not hold (directly or indirectly) by the action of other threads.

1. A thread may put a reference that it holds into the state of a shared object.
2. A threads may put a reference it holds into a message it sends to another thread.
3. A thread may bind the reference to a shared data-flow variable.

In the rest of this chapter we will go into a little more detail on the three sharing models.

12.1.1 Object-oriented sharing

We first look at object-oriented programming languages/systems. Except for code the only types of entities that threads share directly are objects. Other types are shared indirectly. Instance variables of shared objects may be used to communicate values (e.g. integers, strings or object references), where one thread sets the instance variable and others read it. Threads create and manipulate local objects but may any time also share the object reference with other threads via an already shared object.

Synchronization between threads is achieved by locks, or from locks derived mechanisms (e.g. synchronized methods). When one thread holds the lock (or runs a synchronized method) of a shared object, other threads attempting to take the lock are blocked until the lock is released (or until the synchronized method call terminates).

[1]Unbound variables can also be bound to another without taking a value and this merges the value to be, i.e. when one is bound to some value the other becomes bound to the same value. Usually threads wait for variable to be bound to a non-variable, e.g. a data structure.

The prototypical example of an object-oriented concurrent system is Java with threads. In Java threads also share all classes via the name-space. Unlike object sharing there is no need to explicitly pass class references from thread to thread within a process. In (centralized) Java when a new thread is spawned (in the same process) the parameters to the spawning call provide the initial sharing environment (except for classes as described).

12.1.2 Message-oriented sharing

Message-oriented centralized programming systems are not common, but there is one good example, and that is Erlang. Erlang is sometimes presented as a first-order functional programming language. This is arguable but this need not concern us as we are interested in concurrency and not the internal computations performed by a single thread/process. In Erlang the concurrency is on the level of processes, and not threads. The processes do not have shared memory. The Erlang processes should not be confused with operating system processes, they are more fine-grained, and there can be many Erlang processes within one operating system process.

Ignoring code-sharing (and one other mechanism that is described in section 12.1.5) the only interaction between processes is via message-sending. Each process has an associated mailbox, where processes may access messages directed to it. The data structures in Erlang, and hence message contents, are all stateless. They may be records, primitive types, or references to processes. Threads may suspend on its mailbox (receive statement), and wait until it receives a message or alternatively until it receives a message of a certain format or pattern.

All code is implicitly shared (via a namespace) much as described for Java-based systems. Messaging in Erlang is geared to many-to-one and one-to-one communication. There is no intrinsic support for one-to-many or many-to-many communication.

12.1.3 Data-fbw sharing

In data-flow threads synchronize by blocking on non-availability of data. A limited form of data-flow is present in message-oriented systems where threads block on non-availability of messages. More general data-flow is achieved through the use of logical variables or futures. Logical variables are known from logic programming languages and concurrent constraint programming languages; futures originate in functional programming languages. They have also been called single-assignment variables.

The basic idea of the data-flow variable is to separate the act of creating a placeholder for some given data from the generation of the data itself. As data-flow is the least common of the three paradigms we consider a producer/consumer example. One thread produces (calculates) data that is consumed (used) by another.

At some point in the program a complex data structure is to be created with two fields, $struct(A\ B)$. At this point the values of the fields A and B are not yet known, e.g. have not yet been calculated. A and B are data-flow variables. In due course one by one the values of A and B become known and the values are filled in, i.e. the

data-flow variables are instantiated. In the course of time the structure is evolving, $struct(A\ B) \rightarrow struct(1\ B) \rightarrow struct(1\ 2)$. At first both values are unknown, later the value of A becomes known to be the integer 1, and finally the value of B becomes known to be the integer 2.

The reference to the created data structure may be passed to other threads at any time. These other threads may then access the structure. If and when the accessing thread needs the value of a data-flow variable the value may or may not be available. If not the thread suspends, and waits until the value becomes known. This is data-flow synchronization. In our example assume that the recipient thread attempts to access the data structure when it is in the state $struct(1\ B)$. We consider two cases. In the first case the thread first attempts to access the 2nd field B. The thread then suspends waiting until B becomes known. In the second case the thread successfully accesses the 1st field, and continues executing. Later the thread needs the value of the 2nd field, which may or may not be available depending on the relative rates of progress between the two threads.

There are two varieties of data-flow in programming systems. In the weaker form, which we will call explicit data-flow, there is a programming distinction between accessing a *future* and accessing a normal field. Accessing a future is done by a special program construct, which is more expensive and avoided if not needed [83]. In the stronger form, implicit data-flow, there is no programming distinction whatsoever on the consuming side. Suspension takes place whenever the value is needed but still unavailable. For instance, consider the course of an ordinary if-statement that compares the value of a still uninstantiated (free) data-flow variable with a given value. Without knowing the value the proper branch of the if-statement cannot be taken, so the thread suspends.

Our example illustrates an important advantage of both forms of data-flow, that consuming threads are subject to less delay. Without data-flow the producer would have to wait until both A and B were known before passing the reference to the consumer, even though the consumer may make some progress with only the value of A.

Implicit data-flow has two advantages over explicit data-flow. One advantage is that the consumer may be programmed the same way irrespective of how and when the data becomes available, so that program changes on the producing side do not necessitate program changes on the consuming side. Another advantage in centralized systems is that data-flow synchronization between threads is actually less expensive than synchronization via locking.

Examples of data-flow concurrent programming systems with implicit data-flow are KLIC (GHC) [44] and Mozart (data-flow part). Explicit data-flow (futures) exists in a number of programming languages (e.g. SML [126]).

12.1.4 Oz or Centralized Mozart

Oz (or centralized Mozart) is a multi-paradigm programming language that supports all three sharing models.

Message-sending is achieved by the use of the programming construct *port*. A port acts as a mailbox reference for senders. Each port has associated with it a *stream* where messages arrive, in FIFO-order for messages sent from the same thread, but otherwise without any guarantees of ordering. Port references may be held by numerous threads and the messages they send to the port are then interleaved on the stream: this allows for many senders. The stream reference may also be shared between threads: this allows for many readers. Thus all forms of message-sending, one-to-one, one-to-many, many-to-one, and many-to-many are provided for.

Data-flow is implicit and consumption (i.e. reading data structures) can be programmed independently of production. This is not only convenient but also makes for fewer programming errors as compared to programming with explicit state (i.e. objects). With data-flow threads synchronize naturally and automatically on data availability. The object-oriented style is characterized by the difficulty in achieving the exact correct balance between insufficient synchronization which gives rise to race conditions and over-synchronization which gives rise to deadlock.

Experience in Mozart shows, as described in [129], that a good programming rule is to use shared state (shared objects) as sparingly as possible. In many applications it is not needed at all. Mozart does provide for objects and these may be shared and the object-oriented concurrency programming style is thus available when and if needed.

In the object-oriented part of Mozart classes (as well as procedures and functions) are first class values, and can be created, manipulated, and passed by reference within the programming language. Sharing a class is thus achieved in the same way as sharing of objects. The initial sharing environment upon thread creation is typically given by the rules of lexical scoping. The thread does therefore not necessarily have access to all classes defined within the program, and cannot therefore necessarily create a new object of arbitrary classes. Of course, in the centralized setting classes can be given global scope giving all threads access to all classes defined within the program.

12.1.5 Other forms of thread interaction

Many systems have an additional mechanism whereby threads can interact. Unlike the interactions associated with the three sharing models these are very seldom used except in the context of exception handling and failure.

These mechanisms allow threads/processes to monitor and control one another on the level of the thread/process. In Mozart this is the province of the thread library. The system offers thread identifiers. These can be used to enquire about the status of a thread (running, runnable, suspended, etc.) and can be used to stop a thread and inject exceptions into a running thread. In Erlang there is a similar mechanism that links the fine-grained processes.

The main use of these mechanism is for error discovery and handling. In this way cascading process/thread failures may be programmed. Runaway threads, possibly due to programming error, possibly to incorrect indata, may be stopped. Monitoring threads can determine that other threads are in some strange state, e.g. infinite loop.

We do not consider these mechanisms in any great detail. They are not part of the core of concurrent programming.

12.2 Discussion

It is fairly obvious that the three sharing models and the associated programming constructs are conceptually useful. A multitude of real-world concurrent activities are naturally modeled with one of these sharing models. Asynchronous message-passing has its real-world counterparts in the human sphere of e-mail and snail-mail (usually many-to-one), flyers and tv/radio media (both one-to-many and many-to-many). Banking systems use shared state (objects). Even such an everyday activity as people passing through a turnstile can be thought of shared state with a locking mechanism based on sight preventing more than one person trying to use it at the same time. Delegation in collaborative work makes use of single-assignment variables, each person or agent is responsible for some part of a larger whole. Each subtask should, of course, be done exactly once. Very often the natural latency tolerance of the model is used and some progress can be made even when some subtask is not yet complete. Indeed, human activity is often organized to minimize the disruptions caused by a slow person/agent and the only stoppage allowed to occur is data-flow synchronization: it is just not possible to make any progress until the relevant information has been provided as opposed to always waiting for all subtasks to be completed before proceeding to the next step.

Chapter 13

Necessity of Three Sharing Models

13.1 Introduction

In the last chapter we saw that there were three different sharing models for concurrent programming systems: object-oriented, data-flow, and message-passing. Most programming languages/systems provide only one of these as a *primitive*. The exception is Mozart/Oz which provides all three as primitive, i.e. directly supported by the runtime.

Now, in a *centralized* system a case may be made that it is unnecessary to provide more than one sharing model, as the others may be programmed. The ones that are not provided directly by the implementation may then be made available as abstractions on the (true) library level. Clearly this makes for a simpler implementation.

This argument largely holds in so far as expressiveness is concerned. If we limit ourselves to explicit data-flow, ignoring implicit data-flow, it is relatively straightforward to program any one sharing model using the constructs of any other. This will be shown in the following subsections. However, the programmed abstractions do in some cases carry significant performance penalties, as will also be shown. We will also show that implicit data-flow is not programmable within the object- or message-oriented sharing models.

The point that we are making here is that a case may be made that you actually do need to provide all three sharing models as primitives in the centralized case. When Oz was originally developed in the early 1990's the design choices were motivated exclusively by the need of centralized concurrent programming. This was our starting point; it is useful to provide for all three and the greater implementation complexity can be dealt with.

In the following chapters we discuss Mozart and other distributed programming systems. There the argument for needing all three sharing paradigms is much stronger. Programming one sharing abstraction in terms on another (when possible) will impact performance in the context of distribution directly. Primarily, the number of network hops increases. Also the partial failure model will become complex. It is our view, that whatever position one takes regarding the balance between the usefulness of providing all three sharing models as primitive versus greater implementation complexity, that

when distribution is taken into account **the balance must invariably shift to providing all three**.

13.2 Message-sending in object-oriented systems

We begin by considering object-oriented systems and consider how message-sending can be programmed.

To achieve message sending in object-oriented systems (e.g. Java-based) we program a mailbox class. A mailbox object is created with the two public synchronized methods put and get. The sender uses the put method, the receiver the get method. The receiver will suspend if the mailbox is empty, otherwise the earliest received message will be returned. Internally the mailbox object maintains a buffer of received but not yet read messages.

Note that the put and get methods are synchronized. Only one thread at a time may work inside the mailbox object and its associated buffer object. Other threads will suspend and then allowed inside the mailbox one by one in order. This is necessary to avoid potentially harmful race conditions whereby messages could be lost.

The programmed abstraction described above does still not accurately reflect true message-sending. In message-oriented systems, message-sending is asynchronous. In a sense the sender merely puts his/her message in the mail and then continues. In due course, without any explicit action by the sender or the receiver, the message will arrive in the receiver's mailbox. In object-oriented systems method invocation is synchronous, so in the Java program the message will be both sent and placed in the mailbox in the put method call. A more faithful encoding of asynchronous message-sending would be to spawn a new auxiliary thread, give the message to it, and let that thread invoke the put method.

The more asynchronous version is seldom used in centralized object-oriented systems of today, as the cost of thread creation is high. This is, of course, not a language issue but a system implementation issue. Another alternative, which avoids continuous thread creation, is to give each thread an buffer for outgoing messages (in addition to the mailbox for incoming messages). Then an auxiliary thread is created for each ordinary thread from the start, the task of which is to take messages (in order) from the outgoing buffer and place them in the appropriate receiver mailbox.

Support for one-to-many or many-to-many communication is also programmable. Each recipient group is mapped to a group mailbox object. Based on the mailbox abstraction described previously, the group mailbox has new methods where recipients can be registered and deregistered. The group mailbox object maintains a buffer that contains references to the individual mailboxes of the members of the group. An auxiliary thread serves the group mailbox object, taking each received message and sending it to each and every group member.

Both the synchronous and asynchronous message-sending abstractions are more expensive by some small constant factor than when message-sending is directly supported by the implementation or virtual machine. (In most implementations the con-

stant factor is much larger than necessary due to heavyweight thread implementation).

13.3 Data-flow in object-oriented systems

Explicit data-flow is also programmable in object-oriented systems using much the same technique as was used for message-passing. An object is designed to hold the data-flow variable. Instantiating the single-assignment variable is analogous to the put operation in message sending, except that the data-flow variable can only be bound once. In unification-based data-flow systems subsequent instantiations are either no-ops, when the variable is bound twice to the same value, or fails [1] Reading the variable is analogous to the get operation in the message sending abstraction.

Once again there are some (constant factor) performance advantages in making data-flow primitive in the implementation. Note that implicit data-flow cannot be programmed in object-oriented systems.

13.4 Objects in message-oriented systems

In pure message-oriented or data-flow programming languages/systems there is no notion of shared state. More precisely, there is no notion of state at all. From the performance point-of-view this is problematic in single-threaded as well as multi-threaded applications. The lack of state introduces a performance penalty that is on the level of algorithmic complexity. With a stateful data structure such as an array an update of a single element can be done in constant time. Pure declarative languages lack stateful arrays. Instead the programmer works with stateless tuples (records). The corresponding declarative counterpart of the update is to create a new tuple identical to the old one except in one field - the updated one. While this may be logically satisfying the cost of the update is now linear in the size of the array.

However, as noted before, most declarative programming languages/systems today are not pure and generally provide for state at some level. Often (though not in Mozart) stateful programming constructs are not smoothly integrated into the language/system. In Erlang, for instance, there are process dictionaries. This provides for state on the level of processes, but not in smaller scopes (visible only in subparts of the process or process code) or larger scopes (visible to several processes).

Keeping in mind the drawbacks associated with lack of state within a single thread we now turn our attention to shared state (or shared stateful objects). In message-oriented programming processes/threads can be made to serve as shared objects. Invocations on shared objects are mapped to sending messages to the shared process. The shared process serves the request and then sends a message containing the return value back to the sender, indicating that the invocation is finished. Clearly, programming the abstraction rather than making it primitive will entail a small constant factor

[1] This is simplified - complex data structures can also be unified and when the structures are compatible unification proceeds by attempting to unify all the subterms, recursively (see paper 2 7).

performance penalty. In passing, we also note that when the units of concurrency are processes (e.g. in Erlang) that much copying will need to be performed on crossing process boundaries.

13.5 Message-orientation in data-flow systems

A data-flow (logical) variable can serve as a stream, a communication channel, between threads. The receiver can be programmed as a recursive procedure that waits for messages on its end of the stream. In this way one-to-one and one-to-many communication is easily and efficiently programmed.

However message-orientation generally allows for many-to-one (and possible many-to-many) communication. In pure data-flow systems (like KLIC [44]) this can only be achieved by use of explicit stream merging (e.g. binary merge). Although this can more or less be hidden to the programmer, the merging operation is not without cost. Sending a message is no longer a constant time operation but logarithmic in the number of receivers. Also the merge network consumes space (even when not used). Note that this space consumption has no counterpart in message-oriented systems.

13.6 Objects in data-flow systems

In pure data-flow systems (like KLIC [44]) shared objects can be programmed. The most straightforward way is to first program a message-oriented abstraction (as above), and thereafter a shared object along the same lines as for message-oriented systems. There is, of course, a serious performance penalty attached, both for the merge network and also for the lack of state.

13.7 Data-flow in message-oriented systems

Explicit data-flow is easily programmable in message-oriented systems. A dedicated process represents the data-flow variable or stream. Consumers register themselves in the process and then wait for an appropriate message. The producer of the value will send a message to all registered processes. This is more expensive than in data-flow systems by some small constant factor when threads are the unit of concurrency. When processes are the unit of concurrency, like in Erlang, there is an additional copying cost.

13.8 Implicit Data-Flow

One variety of the data-flow sharing model, implicit data-flow, is not programmable in terms of other sharing models. The pros and cons of implicit data-flow as compared to explicit data-flow in a centralized system are largely outside the scope of this thesis.

Nevertheless we will mention some that are useful in centralized programming but even more so in distributed programming.

Implicit data-flow makes it possible to change the pattern of production of data in producer components without changing the consumer components. Sometimes you may want to change the consumer code but this is to increase the latency tolerance (i.e. performance) that the data-flow variable provides by rearranging the accessing order in consumer threads. This is useful in centralized concurrent applications where the timings involved in the production of data are poorly understood or difficult to model. (Clearly the uncertainties of data production timings and ordering will grow when threads are distributed onto different machines).

The logical or single-assignment variable is useful for synchronization. Waiting on data availability is not only natural, but also robust in the sense that an oversight on the part of the programmer which gives one thread access to a partially uninitialized data structure does not break the program. In a distributed context disassociating the act of determining that a value should be produced from actual determination of the value gives latency tolerance. This is described in papers 1 and 3 (chapters 6 and 8).

Another useful property of logical variables that can be emulated in explicit data-flow constructions but is usually not, is the fact that unbound variables can be bound together or merged. Whatever value the variable is eventually given will be visible to threads that reference one or the other of the original variables. An example would be that a server is queried twice for the same information, e.g. the server receives the query(Q A) from the client where Q is the query and A is the answer - an unbound variable. The server then spawns a computation (thread) to provide the answer; this thread will eventually bind A. On the second query the server receives an identical query, query(Q B). The server then needs merely to bind A to B.

13.9 Conclusion

In summary, we have seen that there are performance advantages in making the three sharing paradigms primitive in centralized but concurrent systems. All the paradigms were mutually programmable, if we equate data-flow with explicit data-flow. However, we all saw that there were performance penalties involved in programming one sharing model in terms of another.

In some cases the performance penalty was very serious, i.e. on the level of algorithmic complexity. These were:

- Message-oriented sharing programmed in data-flow languages
- Object-oriented sharing programmed in data-flow languages
- Object-oriented sharing programmed in message-oriented languages

In other cases the performance penalty was less serious, involving some small constant factor.

- Data-flow in message-oriented systems
- Data-flow in object-oriented systems
- Message-oriented in object-oriented systems

Comparing the single–paradigm systems the limitations of both pure message-oriented and data-flow systems are the most severe. Not surprisingly, there are very few pure message-oriented or data-flow systems in use today. Often there is some provision for state (objects).

Of the single paradigm systems only the object-oriented systems come close to meeting our control requirements. Message-orientation and explicit data-flow can be programmed, though there is a small constant factor in performance penalty. It can be, and is, argued that this performance penalty is small and not significant for concurrent centralized programming.

We can conclude that concurrent programming systems that offer all three sharing paradigms, e.g. Mozart, are best from the control perspective. In addition, Mozart also provides for implicit data-flow, which is not programmable in the other paradigms, so Mozart is also best from the abstraction perspective.

We will reexamine the virtues and drawbacks of the various sharing models again when we consider distributed systems. We will reconsider the consequences of programming one sharing model in terms of another in the context of distribution. We shall see that things change and the arguments for providing all three sharing models becomes stronger. In this section, we have seen that a case may be made for the usefulness of providing all three sharing models (as primitives) already in the centralized case.

Chapter 14

Distributed Programming Systems

In this chapter we begin to look at the three qualities of programming systems, abstraction, awareness and control for distributed programming systems. More exactly we discuss the new aspects that need to be addressed for distributed programming systems. Distributed programming subsumes concurrent programming which subsumes sequential programming - so all that has been said about such programming systems will still apply.

Here we consider these qualities on an abstract level. A more detailed description and comparison of distributed programming systems will be undertaken in the following two chapters.

14.1 Abstraction

14.1.1 Transparency

Distribution transparency (network transparency) makes for good abstraction in distributed programming systems. In the distributed case we now have threads/processes running on different machines as well as on the same machine. It is clearly advantageous to be able to abstract out this difference as much as possible when programming. The model of thread/process interaction is independent of thread/process location. The most important practical consequence, if this goal can be realized, is that applications will not have to be designed for any specific distribution structure of threads. For example, consider an application with three threads A, B and C. In one scenario thread A and B run on one machine and thread C on another while in the other scearnio thread A runs on one machine and B and C on another. The program is the same.

Clearly, a distribution-transparent distributed programming model will subsume a concurrent programming model. We need only consider the special case when all threads/processes run on the same machine. Note that the argument for distribution transparency does not say that the subsumed concurrent programming model is necessarily identical to any existing concurrent programming model. Clearly, the subsumed concurrent programming model must still be reasonable with respect to the qualities of

abstraction, awareness, and control. However, the distributed programming model may be richer than a model designed exclusively for centralized concurrent programming.

Total distribution transparency is, however, depending on your point-of-view either not possible or not desirable. We need to consider failure, or more particularly partial failure. In the centralized concurrent case if the process crashes the entire application fails, or put another way, the threads all fail together. In the distributed case a process crash may in general cause a subset of threads to fail. More generally, in a distributed application, there is a difference between all the scenarios where threads A and B both run on the same machine, and those where they reside on different machines. In the first case, the threads A and B are inexorably joined and live and die together. In the second case, one may die while the other lives.

We see that different distribution structures cause different kinds of partial failure situations and these are not distribution transparent. The only way to achieve a kind of distribution transparency here would be to attempt to fail the entire application upon the failure of any one machine. Even if this were doable it is clearly seldom desirable as applications that stretch over a large number of machines would then be extremely vulnerable to failure.

The only remaining alternative is to reflect this failure up to the programming level, where there is the option of either coping with the partial failure or possibly aborting/killing the entire application. Coping with partial failure entails fault-tolerance and programming systems may vary considerably in the support that they provide for building fault-tolerant applications (see also section 2.4).

The situation is further complicated by the fact that failures cannot always be reliably detected on WANs (the Internet). In many cases all a single machine or a thread can know is that there is some communication problem and that the desired interaction with a remote thread cannot currently be performed. Perhaps interaction will be possible later, perhaps not.

The conclusion here is, that to achieve good abstraction distributed programming systems should meet two criteria. Firstly the system should be distribution transparent modulo failure. Secondly, the subsumed concurrent programming system (threads/processes on the same machine) should also exhibit a good level of abstraction judged by the same criteria as pure centralized concurrent programming systems.

14.1.2 Reference bootstrapping

In the chapter on concurrent programming systems we saw that threads interact through shared entities.

Two threads that share the same stateful language entity can potentially interact through the shared entity. We say that they are in the same *sharing domain*. The *extended sharing domain* is the transitive closure of threads in the same sharing domain. Any two threads that belong to the same extended sharing domain but in different sharing domains can potentially interact. For example, threads T1 and T2 share object O1 and threads T2 and T3 share object O2. Thread T1 and T3 are in the same extended sharing domain but not in the same domain. If thread T2 puts a reference to object O2

in O1 this will put threads T1 and T3 in the same sharing domain.

The important point here is that threads that are not in the same extended sharing domain will never be able to interact. This gives rise to a bootstrapping problem, how to give a thread the appropriate initial set of shared references. In centralized programming, ignoring shared namespaces, which are problematic to extend to distribution, threads are given an initial set of references by the thread that created it, and so on. This can be ultimately traced back to the *original* thread, which has full control and can plan for proper initialization of all of its descendants, recursively.

In the centralized case there is no need for threads that end up in different extended sharing domains to ever connect. In the distributed case there is. Two threads are created on two different machines independently need to bootstrap their shared references. Distributed programming languages/systems thus need a new way to connect disparate computations.

This illustrates that to be useful an integrated distributed programming language/system will need to be strictly richer than its concurrent core.

14.2 Awareness

Before we consider the performance model of distributed programming systems, we will briefly consider, once again, the performance model for centralized (possibly) concurrent programming systems. The abstractions that centralized programming systems provide are ultimately translated to machine code, either directly or through the action of low-level interpretation (virtual machine). Associated with the programming system is a performance model that can be used to predict the cost associated with the execution of the abstractions. The cost given by the performance model may be seen as rough estimate of the number of machine instructions that are required to execute the abstraction, somewhat augmented by an awareness of cache behavior, and hardware registers.

In practice such a performance model is too cumbersome and imprecise to predict the performance of non-trivial programs. This is not only because that actual performance will depend on the specific hardware that the program will run on. The sheer size and complexity of most applications makes predicting the performance of a program developed from scratch based on such a fine-grained model impractical. (On the other hand, it may give a good indication where to performance tune existing software).

Instead, the performance model is mostly used to judge the relative performance of different programming constructs available to the programmer. Indeed, the programmer is not necessarily consciously aware of using a model and if asked, why he/she chooses one construct over another, it will be put down to experience. The model is most often used in the small, on the level of the procedure and subcomponents, comparing one possible implementation with another.

Another clear advantage of a relative performance model is that this makes the model more or less hardware independent. Better hardware will make for faster program execution, but the relative merits of different solutions remain roughly the same.

The absolute performance difference when moving from a slow to a fast processor may be difficult to predict, memory-intensive applications do not speed up as much as computation-intensive applications.

As distributed programming systems subsume centralized ones, the model of performance will include all that which is important for centralized systems. The programmer will need this for all purely local computation.

But the performance model for distributed programming will need to contain much more. The abstractions, in general, will ultimately be translated into both machine instructions and messaging. Abstractions may now involve sending messages, receiving messages, and waiting for the receipt of messages.

There are many variations in network capacity and speed (e.g. LAN, WAN, wireless). To some degree this is analogous to variations in hardware for centralized programming. Machines vary in clock frequency, cache size and characteristics and main memory size to name but a few.

However, the spectrum of possibilities is much wider and absolute performance much more difficult to predict. If the application needs to send large messages between machines and then wait for a response, there will be a huge difference in absolute performance if the connection is a low capacity modem rather than a dedicated high speed LAN.

Just as was the case for centralized programming, we need a relative performance model to weigh different constructs against each other. However, we now have three different factors to consider.

- Local computation
- Latency (or number of network hops)
- Message size (bandwidth usage)

Interaction between remote threads will require messages to be sent. Certain kinds of interactions (e.g. synchronous) will typically suspend the initiating thread until the interaction completes. A deployment independent measure of the latency is the number of network hops (two for simple RPC). Even if the operation is asynchronous the number of network hops is important as there may be a thread at the other end which is suspended waiting for the message to arrive.

One aspect of message sending is the size of a message. The larger the message the more bandwidth is required. In general, when messages are large enough (or network capacity is low enough), the time it takes to send a message is proportional to the size of the message. A rough appreciation of the size of the messages is thus clearly part of the performance model of the abstractions.

Compared to the centralized performance model there are now three factors to consider, rather than one. Also absolute performance varies more with changes in deployment (networks or hardware). This makes for a more complicated model. But in many cases not unduly so. For instance, when different constructs are compared to one another, if one construct is better than another in any one of the three factors and

equal or better in the other factors, it is to be preferred. So, sometimes the choice is simple.

However, it is now also necessary to consider situations where there are tradeoffs involved. How do we choose between constructs where one is better in one factor, and the other in another? How do we weigh the different factors against one another? Indeed, how common are these kinds of difficult choices? Clearly systems with good awareness will provide the model to answer such questions.

14.3 Control

The performance gap of a distributed programming system reflects the cost of abstraction in non-optimal local execution, and non-optimal message sending. Non-optimal local execution is when more (or more costly) machine code instructions are executed than could be achieved by direct encoding. Just as the case for centralized programming systems we are probably willing to pay a small constant cost for the ease of programming at a higher level of abstraction. The only really new factor here is that local execution will also encompass serialization or marshaling, i.e. the conversion of data structures (including code) into a serial representation for sending over the network. This needs to be reasonably efficient as well.

Non-optimal message sending is when a gain of performance could be achieved by direct encoding of an application-specific protocol at a low level. The potential non-optimality here is related to the two other awareness factors specified earlier, the number of hops and message size. In one case, the performance gap is seen when thread interaction is ultimately translated into more messages and/or more network hops than in strictly necessary. This increases latency unnecessarily. In the other case, the messages are all necessary in order to achieve the desired thread interaction but the messages themselves are unnecessarily large and bandwidth demanding. This would be the case, for example, when the serialized data-format is unnecessarily bloated or verbose (e.g. XML).

A small performance gap may, as in the centralized case, be acceptable, but we still expect that in time those distributed programming systems that gain acceptance for general-purpose distributed programming will have a very small performance gap. We also believe that minimizing the number of network hops is both very important and, as shall be seen, ultimately also realizable, so that the acceptable performance gap will come to be zero. We also expect that, ultimately, the data formats that survive (for machine-to-machine communication) will be very compact, and that the acceptable performance gap here is very low (not more than on the order of 10-20%).

14.4 New Abstractions and Old Assumptions

It is both natural and straightforward to design and build distributed programming systems by extended centralized concurrent systems. Similarly it is natural to design distributed programming systems for WAN environments by extending LAN-based

distributed systems or shared memory multiprocessor systems. However, this approach is also inherently dangerous.

The dangers are two. The first danger is that the assumptions and invariants of the base systems (before extension) are not accounted for sufficiently. One needs to be very aware of the invariants that have in the course of time been interwoven into the design of such systems. We look at some abstraction of the base system and say - such a nice abstraction we must immediately extend it to distribution or extend it to WAN-based distribution. We forget that associated with the abstraction are control and awareness aspects that may rely heavily on invariants that no longer apply.

The second danger is oversimplification. New abstractions will be needed and dogmatic reluctance to provide them will have severe consequences. It is useful to shift the perspective and consider the two systems not only from the point of view of extending the base system but also from the point of view of simplifying the distributed programming system. We can then expect abstractions and distinctions that are present and important in the distributed system to collapse in the base system. Two distinct abstractions in the full-fledged distributed programming system, with very different control and awareness properties, may in the base case be completely equivalent.

We will need new abstractions. We have already mentioned bootstrapping where distribution (particularly, WAN-based distribution) introduces the need for a new connection or thread initialization abstraction.

An example of a centralized invariant that needs to be reexamined is the use of a global namespace (typically for code). This is not to say that global namespaces are not useful in distributed computing, but only that they are the wrong *default*. Where limited scope will do, it should be used. The property that a certain piece of code and data only needs to be visible in a limited context should be explicit in the program and not implicit in the programmer's head. Often enough, in applications, this limits the scope to a single machine (or even a single thread/process). This is important as such local entities impose no burden whatsoever on the distribution support system.

Another related example is the primitive distributed programming system consisting of an imperative programming language (e.g. C), threads/processes together with distribution support based on shared memory. These distributed shared memory systems (DSM [122, 29]) have their origin in systems for shared memory multiprocessor systems which were extended for distribution. There are three major drawbacks to this approach (aside from the disadvantages associated with low-level imperative programming languages). First, is that the granularity of the units in the consistency protocols, memory pages, has no relation to units of granularity in the programming model (i.e. data structures). This gives rise to the problem of false sharing, where the distribution support system is working hard to keep data consistent to no purpose.

Another drawback with distributed shared memory is that locality is never assured. All of the distributed programming systems that we focus on (Mozart, Erlang, Java, etc.) have the property that there are entities that are assured to be local to a single machine. This can only be broken by actions taken by the threads/processes on that machine - it can never be broken from outside. All language entities in this happy state can essentially be handled by the runtime as if the system was centralized. In

a programming language with pointer arithmetic the action taken by other machines may suddenly cause previously local data to become shared. This imposes a burden on the runtime.

The third drawback to traditional distributed shared memory systems is the lack of distinction between stateless and stateful data. In depth discussion of this is deferred as this is not exclusive to distributed shared memory. Many distributed programming systems also have this property and this will be covered in detail in the following chapters.

There are, we claim, pitfalls associated with naively extending base systems to distribution. In the following chapters we will see many examples of systems that have plunged into these pitfalls. During the course of the Mozart work we were not able to entirely avoid them either. Some of them we were able to climb out off. But in the section on future work(chapter 17) the astute reader might think, and would be right to do so, that not all the proposed future work reflects things that we never got around to doing.

From the Mozart experience we expect that mature (future) distributed programming languages/systems will be much richer than a purely centralized concurrent programming system ever need to be. There will be many more abstractions and distinctions.

Exactly how new the necessary additional abstractions will be does depend on the concurrent programming language base. The centralized precursor of Mozart, Oz-3, was rich in abstractions and distinctions that other languages lack. These were motivated for reasons that had nothing to do with distribution. Properties like referential safety, lexical scoping, higher-order, distinctions between stateless, stateful and single-assignment data were originally motivated by considerations of programming power, elegance, and defensive programming in the context of centralized concurrent programming. As it turned out all these distinctions were just as important for an entirely different reason - distribution.

Chapter 15

Two approaches to distributed programming systems

15.1 Introduction

It is our view that there are only two approaches to building a distributed programming system. The distinction is in the kinds of program constructs available to the application developer for dealing with distribution, i.e. the non-centralized portions of his/her application.

One approach is the message-passing approach. A centralized programming system is augmented with the ability to send and receive messages between sites, processes, or threads. This is a straightforward approach as it exposes the underlying reality of distributed applications to the programmer. Distributed applications do after all consist of a collection of processes on different machines that exchange messages between them, working together to achieve some common goal.

The other approach is the integrated approach, where the language constructs themselves are augmented for distribution. The rationale behind the name of the approach is clear if we first consider a centralized but concurrent application. Threads reference language entities and some language entities are referenced by more than one thread. These language entities are shared between threads. The distributed programming system is then augmented so that the language entities may also be shared between threads on different machines.

15.2 Message-passing approach

15.2.1 Introduction

In the message-passing approach sites/processes/threads can send messages and receive messages. Messages are sent asynchronously, though the sender may block, or

suspend waiting for a reply. Messages once sent cannot, of course, be unsent [1].

The simplest example of the message-passing approach to building a distributed programming system is to use a centralized programming system together with a TCP/IP library. The programming language will then, in addition to the normal centralized constructs, also contain the socket interface functions; read, write, connect, accept etc. The programmer had better understand the semantics of sockets, and needs, for instance, be aware of the crucial difference between blocking and non-blocking reads and writes. In addition, the programmer needs to be aware of the 50 or so various error codes that the socket functions may return, and have a strategy to deal with them. The programmer also needs, to some degree, have a model of the inner workings of the TCP protocol; in particular he/she needs to have an appreciation of the time-outs that TCP uses.

15.2.2 Messaging Service

As our focus is on Internet applications (wide-area networks) all message-passing will ultimately be based on TCP/IP (or UDP) as this is the provided infrastructure. However, more sophisticated messaging services can be constructed by abstracting over TCP or UDP. If the messaging service is built on top of UDP then the messaging service will also have to manage packet loss, duplication, and provide for some form of flow control.

TCP services require the programmer to explicitly manage the opening and closing of connections. Messaging services can be built that open and close connections automatically. The programmer need then only send and receive messages on logical connections. The error conditions of TCP can be abstracted, and rather than relying on TCPs hard-wired time-outs the programmer can configure or program his/her own.

The messaging service can also manage aspects of resource control. In TCP there are generally limits both on how many connections one OS-process can hold open at any one time and on how much data can be written (or read) in any one OS-call. The messaging service can share the TCP physical connections between the logical connections needed in the application. Also it can handle large messages transparently to the application, buffering where necessary, so that the application will only receive complete messages even if the message took many invocations of TCP to send.

Message-sending may be synchronous or asynchronous. This is mirrored in TCP with blocking and non-blocking sends (writes). Introducing buffering in connection with asynchronous sends could lead to memory overflow problems if not dealt with; the messaging service will need to provide some mechanism for flow control on the level of the application.

In many cases there will be concurrent activity within each site as well as between sites. In the case where different threads send messages to the same receiver, over the same or possibly different logical connections, the messaging service may choose either to use the same physical connection or alternatively different physical connections

[1] We exclude rollback mechanisms for parallel simulation - they are much too costly to be considered for distribution

to deliver the messages. In the former case the logical connections are multiplexed. The advantage of multiplexing is that it uses less operating system resources. There are also disadvantages with multiplexing, at least if the messaging service is built on top of TCP, as additional dependencies between threads are introduced.

15.2.3 Data-integrated message-passing

The messaging service, as described so far, requires that messages be passed to it as sequences of bytes. This makes such a messaging service completely language independent. This does not mean that the byte sequence does not represent structured data that is linearized according to some standard or convention. Obviously this must be the case if the receiving process is to understand the message. For interoperability some standard format (e.g. XML) might be used. For applications where all programming is done in the same language other (internal) formats can be used.

In a message-oriented distributed programming system messages will typically be generated with the programming system first, often as complex data structures. In order to use the messaging service the data structure must first be converted into a sequence of bytes. This is sometimes called linearization and sometimes called marshaling. We will use the term marshaling. At the receiving side the byte sequence is unmarshaled and the corresponding data structure generated.

It is much more convenient to be able to pass the data structures of the programming system directly to the messaging service. This is taking the first step toward integration, as the messaging service now understands the data representation used within the programming system. Marshaling and unmarshaling are no longer performed by the programmer but by the messaging service.

There are two varieties of such a messaging service. Either the messaging service marshals the data immediately or alternatively when the communication channel is free of undelivered previously sent messages. The former type we call an immediate messaging service and the latter type we call a fully asynchronous messaging service.

There are important advantages with a fully asynchronous messaging service. The program may run ahead of the ability of the network to deliver messages so that immediate marshaling of the message will only fill the messaging service's internal byte buffers. Often, data representation within the programming system is more compact than the marshaled equivalent, so if the message cannot be delivered yet, there are advantages in delaying marshaling (within limits, of course, eventually, if the network cannot deliver messages sufficiently quickly, the producing threads/processes must be stopped or slowed).

A complication with a fully asynchronous messaging service is that the message must be protected from reclamation until the entire message has been marshaled and possibly even longer. The message may not be garbage-collected. The messaging service must therefore be designed so as to cooperate with the garbage-collector.

An important issue concerns the nature of the message contents, the data that is sent. The interesting distinction here is not between data and code, but between stateless and stateful language entities. Code in most programming systems is stateless, but

data may or may not be. Examples of stateless data are primitive types, records (in Mozart, Prolog, and many other declarative programming languages), and initialized objects in Java where all instance variables are final. Objects are generally stateful, as are structs and variables in C.

Passing all kinds of stateless language entities to the messaging service is unproblematic. The difficulty is with stateful language entities. The application programmer must be aware of the snapshot semantics of his/her messaging service. Say that the stateful entity is subject to a series of state transitions $S1 \rightarrow S2 \rightarrow S3$. A reference to the entity is given to the messaging service when it is in state S1 and it is sent in that state. If later the sender examines the entity and sees S2 the sender should definitely not come to the conclusion that S2 was sent.

If stateful messages are to be allowed then the messaging service cannot easily be made fully asynchronous. Consider the case where the sending thread/process sends the message with the entity in state S1 and later updates the entity to state S2. The messaging service may send S1 or S2 depending on timing, or even worse may send some intermediate state. Also, if there are other threads that reference the entity the state may change due to the action of other threads during serialization. It can be argued that it is the programmer's responsibility to ensure that such race conditions do not occur.

In any case, allowing stateful entities in messages has two important consequences. Firstly, the programmer must be very aware of the snapshot semantics and carefully synchronize his/her program. Secondly, it puts some hard constraints on the nature of the messaging service.

Examples of messaging services are MPI (message-passing interface) [104] targeting C and C++ and the Java messaging service [2]. Both provide for resource control above that of TCP/IP, and immediate messaging. Additionally, both provide for data integration in various programming languages(C, C++ or Java). Note that in both cases that the centralized programming model is based on programming with stateful entities (structs and objects) while data-integration involves snapshot semantics.

15.2.4 Mailboxes and abstract addressing

Up till now we have considered message sending between sites that know about each other. An important practical consideration for open distributed applications is that this knowledge can also be communicated in messages. If sites A and B know of each other (i.e. can send messages to each other) and sites B and C know each other then site B should be able to communicate its knowledge of site C to site A so that site A can send messages to site C.

The simplest naming mechanism is just an IP-address and port. This does not distinguish between different incarnations using the same port, which may not be desirable, and gives each process/site one single mail-box (or one per port), which may be too limited. Another approach is to introduce some mail-box abstraction into the messaging system. There may then be many mail-boxes within the same process. Mail-boxes that belong to different processes are different (even when one process reuses

the same IP-address and port that a previously terminated process used).

15.2.5 Abstraction, awareness, and control

Clearly the message-passing approach can provide the programmer with good awareness and control. Each message sent requires one hop to arrive at its destination. If raw byte sequences are sent then the programmer has a very clear appreciation of the size of the message and hence the bandwidth requirements. Even with data-integration the system can provide a rough model of the size of the serialized message.

That the message-passing approach gives good awareness and control is, of course, no more surprising than the fact that assembler programming, in the centralized case, also provides good awareness and control. In both cases the level of abstraction is very low, and the mapping to the underlying primitive operations is relatively straightforward.

This brings us to the third quality, abstraction. The drawback to the message-passing approach is twofold. Firstly it offers only a very low level of abstraction, greater with data-integration than without, but still low. Secondly, in the context of almost all programming languages, the programming system as a whole has two very different models that the programmer must understand and work with. One model is the programming language used for programming local computation while the other, message-sending, deals with distribution.

15.3 Integrated approach

15.3.1 Introduction

The integrated approach takes its name from the desired property that distribution is integrated into the programming language. In other words distribution is not handled separately using a totally different model than the one that is used for concurrent centralized programming.

The integrated approach offers a promising alternative to message-passing. Potentially it can provide a higher level of abstraction. We will see that some of the abstractions offered in such systems will involve the action of protocols which will map the abstractions not to a single message being sent but rather to a set of messages carefully coordinated to achieve a desired result. Finally in the integrated approach the abstractions used in local computation are integrated with those for distributed coordination, making for fewer and more generic abstractions.

Another way to look at it is to imagine starting designing a distributed programming language from scratch. This distributed programming language will abstract the underlying machinery, computation within one process and basic message sending between processes (machines). This language will necessarily have abstractions for computing within one process and mechanisms whereby processes/threads can interact and synchronize. Clearly, from such a language a concurrent programming language can

be derived - covering the case where the various computations (threads or processes) reside within the same OS-process. Practical systems based on such a language will surely optimize thread interaction within the same OS-process or same machine. Given that the distributed programming language/system was useful then the centralized concurrent part of such languages/systems would be useful as well. Any well designed general distributed programming language/system should therefore contain within itself a concurrent programming language/system (the concurrent core).

Integrated distributed programming languages/systems therefore subsume concurrent ones. This does not mean that the ultimate distributed programming language/system necessarily looks and feels like a centralized concurrent programming system. It probably contains many more constructs and distinctions than are needed for centralized concurrent programming, just as concurrent programming systems contain strictly more than is needed for sequential programming (e.g. locks). Rather, if the distributed programming language/system is stripped of that which is only relevant for distribution, we are left with a concurrent programming core.

So unless there is still some undiscovered useful thread interaction (sharing) model out there the concurrent programming core of today's and tomorrow's distributed programming languages/systems will be based on one or more of the three concurrent sharing models. We can thus categorize integrated programming systems in much the same way as we have categorized concurrent programming systems. We have:

1. Object-oriented
2. Data-flow
3. Message-oriented

Reference systems that correspond to the three types are Java-based (object-oriented) distributed programming systems (e.g. with RMI), the subset of Oz/Mozart without objects and ports (data-flow) and Distributed Erlang (message-oriented).

15.3.2 Transparency

The main idea of the integrated approach is that the model of concurrency is (virtually) the same for both distributed and concurrent programming. A distributed application is just a concurrent application where threads are partitioned between sites. Threads interact with each other and the interaction is the same irrespective of whether the threads are on the same site or on different sites.

This reflects the quality of abstraction that we find desirable in a good programming system. The same model of thread interaction applies within a site and between sites. This simplifies programming in that there is only one model of thread interaction rather than two, one for threads within one site and one for threads between sites. Moreover the interaction between threads on different sites must necessarily involve many steps of computation, serialization (marshaling), as well as the action of a coordination protocol - a fairly complicated mechanism to implement, all encapsulated in a high-level abstraction.

There hare two properties of thread interaction that cannot be made completely transparent. The first is the time it takes to perform the operation. Timings will invariably change when we go from a local operation to a remote operation. Finally, distribution (or changes in distribution) introduces new types of errors due to network partitioning and the crashing of sites. These manifest themselves in new types of partial failure.

However, modulo these two factors, timings and failure, we want the semantics of the programming language to be unchanged. Ideally, we should not be able to observe any other difference whatsoever. Note that the differences, when maximum transparency is achieved, are exclusively in non-functional aspects.

Integrated systems for distributed programming **all** strive toward transparency. This can be seen in RMI (e.g. Java RMI), its predecessor RPC, Object Voyager, etc. But currently most systems are only partially network-transparent. In Java RMI invoking a remote object where all method parameters are simple types is transparent modulo failure and timing. In this case Java RMI is maximally transparent. However when the parameters are (stateful) local objects the remote method invocation is not transparent, i.e. does not exhibit the same behavior as a local method invocation would.

Given that transparency is desirable, the question arises as to why so many systems are only partially transparent (i.e. have many other non-transparencies over and beyond timings and failure). The answer to this question is complex and involves many factors. We will come back to this question in section 16.7.3.The two most important factors, are (1) that transparency with reasonable control is difficult to implement, and (2) that some programming languages (or abstractions) by their very nature can not both be made transparent and efficient (or in terms of our three programming criteria cannot offer good abstraction and control at the same time).

15.3.3 Partial failure

RMI also illustrates the fact that network transparency does not extend to time nor failure. Remote method invocation will invariably be slower than method invocation on local objects; we cannot expect anything else. Furthermore, remote method invocation may produce an error if the site on which the object resides has crashed. This kind of partial error cannot occur in a centralized system. Centralized systems also crash but this causes the whole program to stop running. In a distributed system, one thread performing a RMI may get an error, while another thread continues to work normally.

A distributed programming language will need to be richer than a concurrent one, and offer new abstractions for dealing with partial failure. In Java RMI the language is extended minimally in comparison with the concurrent core; partial failure is reflected in new kinds of exceptions. It is doubtful if this is enough and good abstraction will probably demand much more of future distributed programming systems. The Java RMI mechanism is reactive only, and faulty remote objects are only discovered upon use. In Mozart the mechanism of watchers provide proactive failure detection which is sometimes useful. In Erlang processes can be linked, so that processes can be informed (or killed) upon the failure of other processes. This is also clearly useful as a partial

failure will generally make the continued action of other processes/threads meaningless. It is a good idea to kill these orphaned processes/threads as early as possible. Note that Erlang chiefly targets LAN-based distributed systems where reliable failure detection is possible.

In our view there is much remaining work to be done on how partial failure is best reflected at the programming level. Outstanding research questions are how to cater for different granularities and unreliable failure detection (WANs), both on the language level and in the implementation. It seems reasonable that, depending on the application, that failure will sometimes be most easily reasoned about on the level of individual language entities (e.g. failed object), sometimes on the level of threads/processes, and sometimes on groups of threads/processes or other coarse-grained abstractions. The ultimate distributed programming language/system will probably need to cater for all these.

15.3.4 Reference bootstrapping

We saw earlier that in centralized concurrent systems it is easy to arrange that newly created threads/processes are initialized with the appropriate set of references. This is no longer the case with open distributed systems. We need an additional mechanism to provide threads with the appropriate initial set of references.

A distributed counterpart of the bootstrapping mechanism of the centralized system would be to spawn a new thread (with given references) on a new machine. Such a mechanism can be very useful. For example, in the Mozart programming system the ability to create threads with an initial set of references on other machines is provided by the Remote library. However in general such mechanisms requires infrastructure support that is neither generally available nor desirable (for security reasons) on wide-area networks. Many distributed applications must be bootstrapped collaboratively.

Integrated systems therefore usually offer a mechanism to connect disjoint extended sharing domains. There are different models for achieving this, but the common factor is that it depends on some facility outside the core language to communicate the reference. As a simple example, a server may exist a fixed ip-address and port, clients may connect to it, and be given an initial shared reference.

In Mozart this is provided by the Connection library. At runtime a textual representation or ticket can be created to an arbitrary language entity in one Mozart process. The ticket is then by external means communicated to another Mozart process (e.g. via an e-mail). Feeding the ticket to this Mozart process will then create a fully functional reference to the original entity. It is part of awareness to realize that those entities that are referenced by tickets are referenced outside the system and beyond the control of the runtime and therefore cannot safely be garbage-collected in the normal way. In Mozart tickets come in a number of varieties. One kind can only be used once. This is a semantic property and the system will refuse a second connection attempt. It also simplifies memory reclamation; once a connection has been made the referenced entity is subject to ordinary (distributed) memory reclamation rules. Other distributed programming systems provide (or could provide) similar mechanisms as described for

Mozart.

Note that in a centralized concurrent programming system there is only one way in which a new thread is created. In our distributed programming system there are now two, the normal way (bootstrapped upon creation) and cooperatively.

15.3.5 Object-oriented

Object-oriented distributed programming systems allow threads on remote machines to share objects. Examples of pure object-oriented systems are Emerald [72], Java with RMI[120], and ORCA[14]. Object Voyager and Mozart are examples of non-pure object-oriented distributed programming system providing as they do, to a greater or lesser extent, other sharing paradigms as well.

All the systems attempt, with varying success, to provide network transparency. Method invocations on shared objects should behave the same irrespective of the locality of the thread that invokes it (modulo timings and failure). All these systems are evaluated in a later section, in terms of our criteria abstraction, awareness and control.

In a distributed system one must also consider the sharing of classes and code and here systems vary greatly in their level of ambition.

Emerald is an older system targeted at closed (distributed) systems and has no provision for sharing code. In Java-based systems threads also share classes (code) via two different mechanisms. First, they may be spawned this way. Secondly, the sharing of an object (a dynamic property) indirectly shares the class. More precisely, threads share a name space that indirectly references the class definitions. Sharing via the name space can be troublesome in open systems as different classes (code) may be given the same name. There is nothing preventing this. Together with the fact that actual code is fetched locally, if possible, this means that the wrong code may be executed.

In Mozart classes (as well as procedures and functions) are first class values, guaranteed to be unique; they can be created, manipulated, and passed by reference within the programming language. Sharing a class is thus achieved in the same way as sharing of objects or any other kind of entity. This makes for a powerful system. For example, by storing first class procedures and classes in objects one can create abstractions for code revision. Code is fetched from the (stateful) object. Once execution has begun with one version it will run to completion. Meta-facilities update the code in the object upon need. With system support for memory reclamation of unreferenced code (just like unreferenced data) older versions will be expunged when all threads executing them terminate.

15.3.6 Message-oriented

The one example of a pure message-oriented distributed programming system is Distributed Erlang. Mozart also provides for message-oriented sharing.

Network transparency in this case means that message-sending constructs behave the same way irrespective of the locality of the recipient thread or mailbox (modulo

timings and failure). Both Distributed Erlang and Mozart (message-oriented part) are network transparent.

Clearly, message-oriented distributed programming systems are the easiest to make network transparent, as the message-passing paradigm is straightforwardly mapped to the basic message-passing low level infrastructure. It could be said that message-oriented distributed programming systems stand in the same relation to the basic message-passing infrastructure as classical imperative programming languages/systems stand in relation to assembler. The level of abstraction is low and the underlying infrastructure is clearly visible.

In Erlang all code is implicitly shared (via a namespace) much as described for Java-based systems. Once again the system targets closed systems and there is no way to dynamically share code, though interestingly enough there are provisions for dynamic updating of code revisions.

15.3.7 Data-flow

Mozart is the only distributed programming system that provides implicit data-flow. Explicit data-flow is provided in Object Voyager. Object Voyager will be discussed in detail later. It suffices for now to characterize Object Voyager as a Java-based distributed programming system that,to some extent, in an object framework incorporates aspects of both message-sending and explicit data-flow.

Chapter 16

Evaluation of the Integrated Approach

16.1 Introduction

There are three important questions that need to be addressed about the integrated approach to distributed programming systems. These questions apply to all three kinds of sharing mechanisms, and any combination thereof.

- Is it useful?
- Is it possible?
- Is it practical?

The first question is the easiest to answer. We will argue that there almost exists a consensus in the community on this point.

The second question is concerned about whether or not integrated general-purpose distributed programming systems are at all possible. The main criteria here is network transparency; the semantics of sharing between threads should be identical for sharing within a site and across sites - modulo timing and failure.

The question of practicality is threefold. First, we consider the awareness aspect. In particular, do the systems provide a reasonable model of performance? Second, we consider the control aspect. Our comparison is with a message-passing system, the assembler of distributed computing. Any distributed application, even if it takes considerable effort, can always be encoded as a message-passing application.

Third, we consider failure. In any distributed application sites may crash and the network may become partitioned. While a centralized application fails in its entirety, distributed applications tend to fail in part. Applications need to take this into account, possibly providing some error-recovery, and possibly terminating the application or parts thereof. The question that must be posed is: does the integrated approach complicate failure handling as compared to message-passing? (Even better, of course, would be if failure modeling and handling were easier in the integrated approach).

In the discussion on the practicality, we will also need to reconsider the question previously dealt with for centralized concurrent systems but now in the context of

distributed programming. Do we need all three sharing paradigms, or can we make do with one or two?

16.2 Is it useful?

There are three main arguments for the usefulness of the integrated approach, if it can be made to work. They are

- Simplicity
- Direct parallelization of concurrent programs
- Distributed computing community experience

The simplicity argument has already been touched upon. A slightly different formulation is that given that the programmer prefers using one or more of the three sharing paradigms for programming concurrent but centralized applications, why should he/she have to abandon this style merely because the threads/processes reside on different machines. It is surely preferable to have the one model rather than two different models, one for the centralized case, and one for the distributed one. This argument even applies to message-oriented systems, without integration the programmer would have two message-passing constructs, one within a site and one between sites. We also note that simplicity is an important factor for improving code quality.

In the integrated approach any multi-threaded application can be easily distributed, and thus parallelized. This merely involves distributing the threads/processes among a group of sites. Care must, of course, be taken so that work is divided in such a way as to minimize the synchronization and communication between sites. With complete network transparency the task then becomes a matter of tuning the parallelized application for performance. In many cases, this tuning is relatively straightforward, with threads that interact frequently preferentially placed on the same site, and with the appropriate choice of protocol. Note that here we are considering parallelizing an already concurrent application, not making a sequential program concurrent (or a concurrent program more concurrent) in order to then parallelize the program - this would call for changing the application logic.

The work we have done is geared to systems for open distributed computing on all kinds of networks, and in particular for the Internet. Classical work in distributed computing has mainly been concerned with programming systems or middleware on closed LAN-based systems, often with hardware or infrastructure support that is not available on WAN. Many of the issues in Internet distributed computing do not arise. The range of applications of interest was limited, e.g. database applications and high-performance scientific computing. Typically, after deployment there would be no further need for code sharing. The application ran exclusively within one administrative domain, and there were no new security considerations.

Nevertheless, the issues that were faced in this classical work, we also face in open distributed computing. The problems that they addressed are a subset of those that

we faced, though in many cases the classical solutions, relying on either invariants or infrastructure that we do not have, are not directly applicable.

In this community, the desirability of network transparency, and other related transparencies (distribution transparency, location transparency, etc) has long been acknowledged. Another term that is frequently used is single-system image.

When we look at existing systems, concentrating on fundamentals, we will, for the most part, ignore some minor limitations of current implementations that could easily be improved. As an example in Distributed Erlang there is a fixed upper limit of 256 processes per application. Obviously, for some applications this is not enough. An example is a collaborative application involving many users where at least one process per user is needed.

16.3 Is it possible?

Mozart does demonstrate that the integrated approach can be made to work for all three sharing models. For two of the sharing models it could be argued that this was demonstrated long before Mozart was developed.

The integrated object-oriented approach was first demonstrated in the Emerald programming system ([72, 71]) as long ago as the 1980's. Emerald demonstrated complete transparency, unlike the more recent Java-based object-oriented distributed programming systems. That an integrated message-oriented approach is possible is, of course, no surprise, as message-oriented concurrency is straightforward mapped to a message-passing architecture. The possibility was also demonstrated in Distributed Erlang [138].

Mozart was the first system to demonstrate the possibility of a integrated distributed programming system with data-flow. This was realized by the variable protocol as described in paper 3 (chapter 8)

16.4 Is it practical - dealing with code

Neither Emerald nor Distributed Erlang is really suitable for open distributed computing as they have no or little provision for sharing code. Code is implicitly shared via a common namespace. Code can be shared when threads/processes are spawned on the same machine or alternatively on different machines but within a shared file system. Either there is no mechanism for connecting sites or alternatively there is no support for sharing code when sites connect. Where a connection mechanism exists the systems lack transparency; where one does not exist (or if we discount it) the systems lack vital functionality.

Unlike Emerald and Distributed Erlang some modern Java-based systems (RMI, Voyager etc.) are targeting the Internet and do need to make provisions for the sharing of code. The basic model for dealing with code is still centralized. Code is implicitly shared in that threads share a common name space, and code is then referenced by name. On top of this basic model an ad-hoc mechanism ensures that code is marshaled

and shipped between sites upon need. For instance, in RMI [90] if the receiver discovers that it does not have code corresponding to a given name it asks the sender for a copy.

In contrast, Mozart has first-class code constructions, i.e. classes or functions. These classes and functions can then be used as parameters in method calls, alternatively put in messages, which the recipients can then use transparently. The beauty of higher order programming languages in a distributed setting is that model for sharing code is the same as that for sharing other values, and with network transparency that the sharing of code between threads is modeled the same way irrespective of whether threads are on the same site or on different sites.

In concurrent programming languages/systems without higher order code is implicitly shared; threads share a common name space. There both advantages and disadvantages with this scheme even in a centralized system. We will argue that the disadvantages will be more heavily felt in open distributed computing.

The disadvantage with the name scheme is the potential of name clashes; what is needed is some means to ensure that all names are unique. This may occur even in centralized programming in programming in the large in projects where there are many programmers that might choose the same name for different functionality. The problem is ameliorated through the use of a hierarchical name space. Sometimes the depth of the hierarchy is limited, sometimes not. Concepts such as packages, modules, etc. are examples of the terms used in characterizing the name space.

In Mozart, classes, functions and procedures have token or pointer equality. Two classes, functions or procedures are only equal if they refer to the same piece of code that was originally generated by some compiler and thereafter replicated. This is supported in the implementation, and name clashes cannot occur.

The other side of the coin is that it is sometimes desirable to give the same name to different pieces of code (different in the sense of having different origin). This is the case with code revision and is an essential ingredient in the concepts of component programming allowing one component to be updated independently of the other. With names this is easy to achieve.

In Mozart there is a distinction between code that is to be used in a component fashion and code that is not. Mozart has the notion of functors, which are referenced and linked by name. In the distributed case when code is sent from one machine to another there is a difference between sending a function or class versus sending a functor. In both cases the receiver may already have all or some of the needed code, but there is a difference. In the first case if the receiver already has the function or class it is guaranteed to be identical, while in the second case the receiver will link in a, hopefully, functionally equivalent (or at least conservatively extended) functor or component. In the second case, there is, of course, no guarantee that the result will work as expected.

In our view, in open distributed computing, the amount of interaction between programs that have been independently developed increases, and thus the problem of name collision is more acute. The ability to create and share code that is guaranteed to be unique is, we believe, very useful.

As a final remark, we believe that there are a number of outstanding issues in distributed computing with components; issues that we have not addressed in Mozart. The classical versioning problem is augmented by the question of locality. Given that a piece of code is referenced by name where should the receiver fetch code referenced by name that he/she does not have? Fetching from the sender is not necessarily the most optimal in a heterogeneous network. Also fetching from the sender presupposes that the sender has the relevant code. If the sender is merely delegating this burdens the sender, as the sender must either store code he/she has no need for, or at the very least store information as to where the code may be fetched.

16.5 Is it practical - awareness

The simpler the protocols that provide for network transparency, the less problematic awareness is. Message-oriented systems have only one protocol, the trivial one, sending a message between processes is done asynchronously and takes either zero or one hop to arrive depending on whether the process is remote or local.

Even in more sophisticated systems, with more complex protocols, many systems exhibit good network awareness.

RMI [90] is realized by a very simple protocol as well and offers good network awareness. Objects are located at the site they were created on. Invocations take zero or two hops depending on if the invoking thread is on the same site or not. The mobile object protocol of Mozart (described in paper 3 in chapter 8) is more complicated than RMI, but also provides fairly good awareness. In general, invoking a mobile object takes 2 or 3 hops depending on where the manager is located. The operation can be performed locally (0 hops) if the object has been cached on the site. This is the case when the object has been invoked previously and no other site has invoked the object in between.

In Emerald objects are also stationary by default, though they may be explicitly moved. The two-hops property in Emerald is only guaranteed if the object is never moved. If the object is moved the invocation may take an arbitrary number of hops as the message representing the invocation moves along the path of forwarding references. This demonstrates poor network awareness as well as increased failure vulnerability.

16.6 Is it practical - dealing with shared state

16.6.1 Introduction

In this section we consider shared state or shared objects. More precisely, we consider shared state where the consistency model is sequential consistency. Sequential consistency is the default consistency model of concurrent programming systems, and at the same time the strongest consistency model that is possible to implement in most distributed systems (i.e. one that does not require absolute global time) [122]. Sequential

consistency also corresponds to our intuition of concurrency in the real world - there is some order in which changes (i.e. updates) occur on an object and successive accesses by an individual (agent or thread) will respect that order.

16.6.2 RMI and Mozart

In Java with RMI and in Object Voyager objects are always stationary [90, 120, 3]. An object is created on a given site, the home site, and is never moved from that site. There may be an arbitrary number of references to the object from other sites, so-called remote references. If a thread invokes a method in a remote object the thread is suspended and a message is sent to the home site of the object. The message contains the name of the method as well as the marshaled arguments. Upon receipt of the message and unmarshaling of the arguments an auxiliary thread on the home site invokes the method on the object. Upon termination of the method call the return argument is marshaled and sent back to the original invoking site. At the original invoking site the message is unmarshaled and the return value is passed to the original invoking thread that is now woken. An important property of the stationary object protocol is that it generally takes two hops for each method invocation (except in the special case of method invocation at the home site).

In Mozart the stationary object abstraction as described in papers 1 and 3 (chapters 6 and 8) works in much the same way (considering only distribution and ignoring differences in the amount of local computation). In Emerald objects are also stationary by default, though they may be explicitly moved. The two-hops property in Emerald is only guaranteed if the object is never moved. If the object is moved the invocation may take an arbitrary number of hops as the message representing the invocation moves along the path of forwarding references.

The mobile state protocol of Mozart is described in paper 2 (chapter 7). Objects are seen to be composed of two parts, a stateless and a stateful part. The class and methods (the code) are in the stateless part. The described protocol deals with the stateful part, which can be moved from site to site, but exists in only one copy. When the object is invoked it may be that the state is already present on that site, in which case the operation is performed locally. Alternatively, the thread suspends and the protocol ensures that the state will eventually arrive in marshaled form in a message. Upon arrival, the state is unmarshaled and the thread woken and local execution of the method commences. An important property of the protocol is that in the worst case it takes three hops for the state to arrive. In the best case the state is already available locally, due to a previous invocation, and it takes zero hops.

16.6.3 Use Case Analysis

We now analyze the performance of various shared object/state implementations. We look at the network aspects, and ignore centralized computation aspects.

We make a number of simplifying assumptions. First we assume that the network is symmetric, or rather that it is not asymmetric in a predictable way. We assume

that latencies are either roughly the same between all sites, or at least varying in some unpredictable manner [1]. We also assume that the object state is generally used in its entirety, or more precisely that we cannot easily partition the state and the invocations such that invocations of different types consistently operate on only part of the state.

We may analyze method invocation patterns from the following viewpoints

- Site invocation pattern: random vs. repetitive
- Update frequency: read-intensive vs. write-intensive
- Number. of active references: many or few
- Numeber of passive references: many or few
- State size: large vs. small

A random site invocation pattern is one where object invocations take place randomly within the sharing group. The contrast is the repetitive pattern where typically one site/thread repeatedly invokes the object before some other site/thread does so. We can distinguish between the read-intensive invocation pattern, where a great majority of object invocations do not change the state of the object, i.e. merely read the state, from a write-intensive one, where updates are fairly frequent. We can also distinguish between shared objects that have many active references, i.e. sites that are actively using the object regularly from those that have many passive references. We may also distinguish between objects whose state size is large versus those whose state size is small. The size measure, here, is the size of the marshaled representation of the state. We assume that the programmer has a model that allows him to roughly estimate this from the nature and type of data structure represented by the object.

Our first observation is that if the state is large enough then it is disadvantageous to move it all, i.e. a stationary object offers the best performance. In the following section we focus on objects where the state is of reasonable size.

16.6.4 Consistency Protocols

Classical results from distributed computing apply here. For instance, one can consider cache coherency protocols. Although the actual protocols for open WAN-based distributed computing will differ due to different assumptions it is doubtful if new fundamental strategies will be found.

The three strategies are:

1. Stationary: the state can be kept in one locality and all operations on it are done at this location
2. Mobile (or migratory, cached): - the state is moved to the place where the operation is performed. The state can be seen as a token - there is only one copy of the state at any one time (globally).

[1] If hops have different but predictable weights this might be leveraged by the protocol.

3. Invalidation-based (or replicated): the state can be replicated freely. Whenever there is more than one copy in the system the copies are read-only. A write operation on the state requires that all copies but one are invalidated.

The third strategy has sub-varieties. One particular dimension that can be important on networks with high latency is if after invalidation and update the new value is propagated eagerly (at once) or lazily (when needed).

If the object invocation pattern is both random and write-intensive then with the stationary object approach there is no performance gap (i.e. this is optimal) . The other properties of the invocation pattern are of no interest here. If the object invocation pattern is random and read-intensive it is better to replicate the state and use an invalidation protocol to maintain consistency. If the number of passive sites is low it would be advantageous to replicate the object state eagerly. If the number of passive sites is high then lazy replication is better.

Let us consider an example. Assume that the frequency of reads to writes is 100:1, we have 10 sites, and a lazy replication protocol is used. Each site will then cyclically access the state 100 times and then update it. During each cycle the state is updated by other sites 9 times. Each access after an update (invalidation) takes 3 hops (to get a copy of the state from the site with the one and only copy). This makes for 3 x 9 hops for access operations. In addition, the invalidation operation during the one update takes 4 hops or less. This is on the order of 30 hops for the entire cycle or 0.3 hops per operation. Compare this to using stationary objects, where 2 hops per operation are needed. Using the mobile state protocol of Mozart would be even worse with up to 3 hops per operation.

If the object invocation pattern is repetitive the situation is quite different. Let us attempt to characterize this invocation pattern. We assume that each site repeatedly does the following: first i invocations within the time interval t, and then does no invocation for time $1-t$. The involved sites are not otherwise synchronized. If we consider one site the probability that during its invocation interval t that no other site interferes by also requesting the state is approximately $(1-t)^N$. The average number of hops per invocation is therefore $((1-t)^N * 3)/i + 3/i$. When t is small in relation to N, the second term dominates and the average number of hops is close to $3/i$.

A summary is given in the table below (where ? indicates any value).

Invocation	R/W Intensity	No. Passive	No. Active	State Size	Best Protocol
?	Read	Many	?	Normal	Lazy Invalid
?	Read	Few	?	Normal	Eager Invalid
Random	Write	?	?	Normal	Stationary
Repetitive	Write	?	?	Normal	Mobile
?	?	?	?	Very Large	Stationary

16.6.5 Conclusion

It is our belief that a system that offers distributed objects must from the control point of view offer all of the above versions of distributed objects (stationary, mobile, and

lazy and eager invalidation-based). For any system that does not offer them **all** there is a performance gap on the level of algorithmic complexity (number of hops). Most distributed programming systems offer only one, Mozart offers two, which is better, but still not good enough. See also future work (chapter 17).

Programming systems with poor control can often be circumvented by moving to a lower level. For instance, consider an application where object invocations are repetitive, programmed using a system with only stationary objects. In particular, consider a Java RMI system. Assume also that the group of sites that share the target object is known statically, as otherwise it is difficult without configuring the class loader. Instead of making the target object remote we could instead keep the target object local (making use of copying semantics). We make use of proxy objects that may or may not hold the correct copy of the target object, and one manager object. Both the manager and proxy objects are made remote. All proxies know their manager. At startup one proxy object has the true and local target object, and the manager knows which proxy this is. Invocation of the proxy that is holding the target object is immediately delegated to the target object. Invocation of other proxies will lead to invoking the manager, which in turn will invoke the proxy holding the target. A copy of the target will eventually arrive back to the calling proxy, changing the state of the proxy, and delegating to the newly unmarshaled target object. Note that given our assumptions this is a perfectly reasonable thing to do. But the solution is a just a poor man's version of our mobile object protocol. It is both less general and less efficient (e.g. fetching the state will take four hops (rather than three).

We should note that in the Mozart system of 2000, stationary objects were an abstraction built on top of objects and ports and was not primitive in the system. The drawback of this is that such stationary objects are inefficient when used locally. This makes Mozart deficient in the control aspect.

As a final remark, there may useful relaxed consistency models for state that we have not considered here. Sequential consistency may not be the only state semantics that deserves to be integrated into a distributed programming language/system. The motivation would be to be leverage the relaxed consistency and use a lighter consistency protocol. We did not investigate this.

16.7 Partially transparent systems

16.7.1 Introduction

Mozart does demonstrate complete network transparency for all three sharing paradigms. Already in the 1980's Emerald [72, 71] demonstrated network transparency for object-oriented sharing and in the early 1990's Distributed Erlang [138] for message-oriented sharing. Despite this most modern distributed object-oriented programming systems are only partially network transparent.

RMI, in Java or Object Voyager [90, 3] is only partially network transparent. The invocation on the remote object may be made in the same way as invocations on local

objects, i.e. is transparent. Also as long as the parameters and return values are primitive types, references to other remote objects, or local stateless objects transparency is ensured. Transparency tends to break down when the parameters are stateful local objects. The local object will be marshaled and a copy created at the caller site. There are now two copies of the conceptually same object where both, in general, may be updated, giving at some later time two inconsistent views of the state of the object. Transparency can therefore only be obtained by very good programmer discipline, e.g. by ensuring that the only objects that will be used as parameters to remote method invocation are objects whose instance variables are all final.

Actually there are other non-transparent aspects to RMI. One important factor is that locks are not reentrant over remote method invocations (but are so over local method invocation), so it is quite possible that the distributed program deadlocks while the corresponding centralized one would not. Note that subsequent to the work presented in this thesis some interesing java-based systems have been developed that are considerably more transparent [10].

The non-transparency of other object-oriented systems is also discussed in paper 3 (chapter 8).

Given that the older Emerald system is network transparent one can ask why more modern and popular systems are not. To answer this we must consider the distinction between stateful and stateless data structures.

16.7.2 Stateful versus stateless

In Mozart there is a clear distinction between stateful and stateless language entities. Object attributes and cells are stateful. Logic variables or single-assignment variables are stateful but they can only change their state once. All other entities are stateless. In particular, records and tuples are stateless complex data structures.

This distinction is the key to efficiency in a distributed setting. Whenever stateless data structures are shared between sites they are safely replicated or copied. There is no need for any kind of a consistency protocol (more or less, see also section 16.8.4).

In object-oriented programming languages objects are generally used for structuring both stateful and stateless data. The distinction between the two very different types of data is either not expressible or alternatively not generally maintained. In Java an object where all instance variables are final is stateless after creation. Programmers may easily leave out the final declaration, indeed, in a centralized setting the only purpose of final declaration is defensive programming - to trap erroneous attempts to update the state. Furthermore, there is no way to express that an object becomes stateless after initialization. The practical consequence of this is that Java programs abound with stateless data structures masquerading as stateful ones.

16.7.3 Java paradox

If Java-based systems were to implement complete transparency then all objects, except for those where all instance variables are final, would upon sharing be made into

shared objects (e.g. remote). All accesses except at the original site would require two network hops. Clearly this is unacceptable if the object is stateless and only masquerading as a stateful object.

Practical use of Java-RMI is based on the programming discipline that stateful objects that will be shared should be declared (i.e. compiled) as remote objects, and those that are stateless when shared should not be declared as remote. The latter masquerading objects will when shared (e.g. used as a parameter in a remote method invocation) be copied, which, of course, is exactly the right thing to do. This is very risky and defensive programming would not be possible. Slightly better would have been to introduce a new operation into the language, that fixates an object - i.e. makes the object stateless. A fixated object can be freely replicated between sites, but, unlike the case with an ordinary local object, attempts to update such a fixated object would result in a runtime error. Non-fixated objects would be automatically made into remote objects when and if shared. Such a system would then be transparent, or at any rate much more so, if lack of reentrant locking is taken into account.

16.7.4 Distributed Erlang

Distributed Erlang, like Mozart, carefully distinguishes between stateful and stateless data structures and is transparent. However in Erlang the kinds of language entities that may be shared are extremely limited. Shared entities come in two basic types, tuples which are stateless data structures and process identifiers which are stateful.

16.7.5 Conclusion

We conclude that the distinction between stateful and stateless language entities is crucial in distributed programming languages. Probably, it is also important that the programming model is such that the programmer is encouraged to use stateless entities whenever possible. The system should provide the right default. By this argument it would be better in a Java-like language where objects alone are the structuring mechanism to invert the default - programmers would have to declare instance variables as non-final. Lack of programming discipline would then result in stateful objects masquerading as stateless, rather than the other way around. But this masquerade will be discovered early in program testing when updates are attempted.

By the same logic, the system should provide abstractions to make stateless data structures out of stateful ones. There are applications where a data structure is stateful in one phase of a program only to become stateless in a later phase. In Mozart there are some useful abstractions associated with the built-in data types, arrays and dictionaries (a dictionary is a dynamic key-value store). These are stateful data structures. The stateless versions are tuples (indexed by number) and records (indexed by name or atom). The system provides abstractions to convert between the two types (or more precisely, to construct the corresponding type with the same contents). Thus, the programmer can capture the property that an entity during its lifetime changes from stateful to stateless.

16.8 Stateless Data Structures

16.8.1 Introduction

In the previous section we saw the importance of the distinction between stateless and stateful language entities. Stateless language entities are just so much easier to distribute. No heavyweight consistency protocols are necessary, structures can be freely replicated between machines. Even this is not trivial as it involves marshaling at the exporting site and unmarshaling at the importing site. Also the story does not end there, as even with stateless language entities there are additional issues to consider.

In Mozart there are also different types of stateless language entities. In the concurrent language base there are two types, while in (distributed) Mozart there are three distinct types in the distribution support subsystem.

In the language there is a clear distinction between language entities that have token (or pointer) equality and those that have structural equality. Classes, procedures, functions and objects (which internally may contain stateful instance variables or attributes) all have token equality. Integers, records, and tuples all have structural equality. The equality type determines the result of equality tests. Token equality means that entities are only equal if they reference the same copy (i.e. derived from the same source).

For many types of language entities only one type of equality makes sense. Only structural equality makes sense for integers. Integers that have been generated at different program points are equal if they take the same value. All non-stateless entities are only ever given token equality. For all but the simplest data structures token equality is the easiest to implement (in a centralized system). Records and tuples are complex data structures with structural equality, and this forms the basis of many useful programming techniques.

Programming languages must define equality for all entities that can be subject to equality tests. In a language like Oz/Mozart, where all entities are first class values, this must be done for all language entities. Interestingly enough, in Oz/Mozart, there is also a built-in data type (chunks) that can be seen as a token equality version of a record.

16.8.2 Implementation of Token Equality

In transparent distributed programming systems care must be taken to preserve equality semantics. In a distributed programming system it is token equality (for stateless data structures) and not structural equality that is the most demanding of the implementation. Firstly, equality must be recognized even when the same data structure is imported at different times and possibly from different sources. Secondly, there are many implementation advantages of making the internal representation of local stateless entities and imported (and hence globalized) stateless entities as similar as possible. For language entities with structural equality the second consideration does not arise. The entity is constructed upon unmarshaling exactly the same way it would have been had it been created locally.

In Mozart stateless entities that have token equality semantics are given globally unique names. A table, called *the gname-table*, keeps track of all such entities that the process or machine knows. The table is only needed and used upon import and export. Garbage collection was augmented to clean-up the table on loss of local reference.

Note, that in Distributed Erlang that the issue does not arise. The only kinds of entities in messages are stateless data structures with structural equality semantics and stateful process identifiers.

16.8.3 Distribution Consequences

An interesting consequence of the Mozart implementation is that, for stateless entities with token equality, Mozart maintains the at-most-one-copy property. Each process/site holds at most one copy (but arbitrarily many references) of the data structure.

This is not the case with structurally equivalent data structures. More precisely this is not the case in the Mozart implementation nor in the Erlang implementation (nor, for that matter, in Java if we consider stateless objects). Consider two threads/processes on different machines that continuously pass back and forth a large tuple or record (say, in a message). This will give good exercise to the marshaler, unmarshaler, and garbage collector. In the case of Mozart it is quite different if the threads are on the same machine as only a pointer will be passed between the threads. In the case of Erlang where all structures are copied between processes there is little difference between the sharing within and between sites (of course, sending messages within one OS-process is faster and simpler than between different OS-processes).

The Mozart implementation (and others) could have been built to preserve the at-most-one-copy property by a mechanism similar to that described for structures with token equality. This would guarantee that there would be at most one copy per machine/process of all data structures with the same source. But is this desirable?

There is clearly a trade-off here as management of the *gname table* (see previous subsection) does cost and costs even when no repeated import actually take place. Maintaining the at-most-one-copy property is only important when both (1) the data structure is large and (2) repeated imports is likely. For large structures the extra table overhead is small compared to the marshaling/unmarshaling overhead.

Interestingly, in Mozart there is a strong tendency for stateless data structures with token equality to be large and for those with structural equality to be small. Classes, procedures, and functors (components) with token equality tend to be large. Integers and atoms are small. Records and tuples (with structural equality semantics) tend to be small. We see that Mozart has the right *default*. But there are many exceptions. For instance, in Mozart there is a programming technique which is fairly common that involves creating anonymous procedures and sending them to other sites. Not only are such procedures typically small but they are guaranteed never to be used in equality tests on the sending site. (Despite this they are handled in the same way as ordinary procedures and are recorded in the gname table).

From the control point-of-view the right default is not enough. The semantic properties of token and/or structural equality is one thing while distribution behavior is

something else. It would be desirable to be able to annotate large records/tuples to ensure the at-most-one-copy property independently of equality semantics. We might also consider augmenting the language to support procedures/classes with structural equality and record/tuples with token equality. (The built-in data type chunk in Mozart has a slightly different interface compared to records - and thus is not quite a record with token equality).

16.8.4 Lazy, eager and immediate

The distribution strategy for dealing with stateless data structures is replication. Furthermore, after successful marshaling and unmarshaling of such a data structure the dependency between sites is broken (i.e. the sender can crash without affecting the receiver).

Nevertheless there are many varieties of this strategy. The first distinction is between eager and lazy replication. Eager replication triggers replication (which involves marshaling at the sending site, sending and finally unmarshaling at the receiving site) as soon as the entity becomes shared. Lazy replication sends only a reference when the entity is shared: replication is only done when and if the receiver attempts to access the entity.

The advantages with the eager approach are straightforward. The data is brought over as quickly as possible to reduce latency and the dependency between the sites is broken as quickly as possible so that the application is less likely to be perturbed by partial failure.

The advantages with the lazy approach are less straightforward. There are two advantages. The first, which is probably less important, is that the lazy approach means that sometimes the whole replication process (marshaling, sending and unmarshaling) is completely avoided. This occurs when the entity is never actually accessed at the receiver site. This does not only happen in exceptional circumstances, large data structures may be shared between sites where some sites access only smaller portions of the structure.

The second advantage to the lazy approach is that this may allow us to avoid running out of memory.We exemplify the second advantage with an example from Mozart which at the same time motivates the chosen default in Mozart. Object in Mozart are complex objects. Only the instance variables (attributes) are stateful. The remaining part of the object is stateless (possibly also including some single-assignment features). In a large distributed simulation application numerous objects are partitioned between sites. After initialization each object is connected (via instantiated features) to a small subset of other objects. The objects form a simulation network and during the course of the running application objects interact with their neighbors (via, for instance, the stateful attributes). In an eager strategy each site would have copies of all objects, as transitively all objects are reachable by following neighbor references. We would quickly run out of memory. (In this case the marshaling/unmarshaling overhead is less significant as the object network once initialized lives for a long time). In Mozart shared objects (the stateless parts) are lazily replicated (and are characterized by the

at-most-one-copy).

For named entities there is a further consideration. By named entities we mean entities that the implementation enforces the at-most-one-copy property upon. In Mozart only the stateless entities with token equality are currently named, but as discussed in the preceding section, other entities might benefit from naming in this implementation sense as well. The *immediate strategy* is to send both the name and the contents upon sharing. In this case the importing site upon discovering that it already has a copy of the entity links in its copy and discards the received marshaled representation of the contents. Another strategy, which we call the *eager* strategy sends the name only. Upon receipt of the name the receiving site will either link in its own copy or immediately request the contents from the sender.

The immediate strategy is the most eager strategy of all and has all the advantages associated with maximal eagerness. The eager (but not immediate) strategy can reduce bandwidth consumption considerably at the price of increased latency and some transient dependencies between sites. Once again, Mozart has chosen suitable defaults for the various kinds of language entities, but the strategy is determined exclusively by entity type. We believe that all the three strategies are motivated in a distributed programming system. Mozart shows the way but is currently deficient in control. The ideal would be to disassociate the distribution strategy from the semantic (language) properties. Stateless entities with token equality, like procedures, classes and functions, could be annotated with an for the application appropriate distribution strategy.

16.8.5 Ad-hoc Optimizations

We should also make clear that the strategies for dealing with stateless entities, as described, are fundamentally different in distribution behavior. For instance, laziness affects the partial failure properties. There are also a number of ad-hoc optimizations that distributed programming systems can make use of to lessen bandwidth consumption without changing the distribution behavior.

In Mozart the marshaler ensures the at-most-one-copy of a data structure in any one protocol message. Atoms in Mozart and Erlang are built-in data types much like a string in Java except that the implementation ensures ensures the at-most-one-copy within an OS-process. Each OS-process maintains an atom table to ensure this. Atoms were developed for sequential programming and have many advantages over strings, for example, they can be compared in constant time. However, when shared between sites, the full string representation of the atom must be sent (each OS-process has its own atom table totally unsynchronized with others). In Distributed Erlang there is, for this very reason, associated with sender/receiver process pair a cache of recently used atoms, in order to be able to use shorthand representations in the marshaled format (in subsequent messages).

16.9 Data-flow

16.9.1 Protocol properties

Data-flow or single assignment variables can from the distribution point-of-view be seen as constrained state. The state is constrained in how it can be updated or changed. The implementation of the associated protocol leverages these constraints to make for a lightweight consistency protocol. True state can be updated many times and this requires a (relatively) heavyweight consistency protocol. This is not the case with single assignment variables. The protocol described in paper 3 (chapter 8) can be compared to the closest related true-state consistency protocol - invalidation with eager replication. Compared to true invalidation protocols the variable protocol leverages the single-assignment constraint and optimizes the eager invalidation protocol as follows:

- No invalidation messages need be sent
- The distribution infrastructure (proxy/skeleton network) can be dismantled automatically and safely after the one and only one eager propagation of the value.

The first point is due to the fact that all readers automatically synchronize (wait) when the variable is still unbound, and the variable is never invalidated. The second point reflects the fact that the distribution infrastructure (i.e. the network of skeletons and proxies) is only needed once.

16.9.2 Constrained State

There are many conceptual advantages of data-flow in concurrent and distributed programming. They are fully described in [129] under the concept of declarative concurrency. Distribution introduces another advantage. The general formulation of the rule is *constrained state should be visible in the programming model*. This is in order to take advantage of the constraints in the consistency protocols. (It is not a bad idea to make them visible from the point of view of program readability and defensive programming as well).

Single-assignment is one such type of constrained state. There are clearly others (aside from the already described stateless ones which can be seen as the most constrained of them all). On the language level we can think of incremental sets and bags (familiar from Prolog). The state is constrained such that the only updates that are allowed are those that add elements to the set/bag. There are clearly many applications that could benefit from support for distributed sets/bags. We will not go further into these possibilities here: this is beyond what we did with Mozart.

However, as we firmly believe that we have contributed to the methodology for building distributed programming languages/systems, we now consider how we want to think about sets in a distributed programming context.

First, we would give them a precise semantics and evaluate their usefulness. (The mathematical concept of sets says nothing about the order in which two independent

observers see elements being added). The next stage, if there is one, is to determine if we need to augment the kernel or base language. At this stage we are concerned with semantics only. For sets and bags , we probably will not need to, we can program a set abstraction and put it into a library. There may, of course, be several alternatives here. Next, this abstraction must be analyzed from the awareness and control aspects. Even though the abstraction need not be primitive in the language it may need to be in the implementation for control reasons. An important part of this analysis is to examine the distribution behavior. It may be that all conceivable library abstractions involve more networks hops, greater latency, or appreciably higher bandwidth use than we know to be necessary. If so, the set/bag entity types should probably be primitive in the implementation. Note that this is exactly what was done with object in Mozart. Objects are primitive in the implementation but their semantics is given by an abstraction making use of cells higher-order procedures.

16.10 Asynchronous versus synchronous

16.10.1 Objects versus message-sending

In message-sending sharing processes/threads interact by sending messages to one another. This is asynchronous and the sending process/thread immediately proceeds with the next instruction. In object-oriented sharing processes/threads interact by method invocations on shared objects. This is defined to be synchronous and the process/thread must wait until the call returns. Sometimes the method will return a value that the invoking thread needs. But processes/thread must wait even when the method does not return any value (e.g. returns void).

We begin by comparing these two programming constructs, message-sending in general versus object invocations without return values. The objects are stationary objects (i.e. the only type supported in most Java-based systems). The main differences are:

- Exception propagation
- Invocation completion guarantee
- More precise failure detection
- Less delay

In remote method invocation exceptions encountered during method execution will be propagated to the calling thread (unless caught). This is not the case with message-sending. In this paradigm the calling thread continues execution and receives no information as to any problems at the receiving end. Of course, later on the sender may deduce receiver problems by lack of expected response, or by the receipt of error-reporting messages. In any case, awareness of the problem is delayed and the sending thread which has continued execution may very well have done unnecessary work.

When the remote method invocation returns the invoking thread is assured that the invocation has successfully completed. This guarantee almost, but not quite, captures the intuition that the receiver has completed all induced activity. It does capture this on the level of the object method call but not necessarily on the level of the programmer's model. For example, if during the invocation new threads were created to handle some of the work then these may still be running when the method call completes.

Messages may be sent to failed machines (or machines that fail before they can process the message). Either way the system provides no information as to failure, or very imprecise information. In general, the sender does not know if the receiver actually ever received the sent message.

Finally, the advantage of message-sending is that a sending thread need not wait. Here we see the trade-off. The trade-off may be summarized as latency (delay) tolerance versus loss of precision about exceptional circumstances.

From the control perspective the system should provide both. The stationary object RMI-paradigm is easily programmed in message-oriented systems. In Erlang the instruction following the send will be a receive statement waiting for the appropriate response. The converse, however, is in practice, more costly to achieve, at least without lightweight threads. The method call can be done in its own especially created thread. In distributed programming systems like Mozart that offer both sharing models both constructs are commonly used.

The above argumentation can be generalized to reason about method invocations that do return values but those values are not immediately needed. Note, that if the value are needed immediately then message-sending systems would need to proceed to a receive statement directly after sending the message, and would thus emulate synchronous method invocation.

16.10.2 Object Voyager

Object Voyager [3] is an interesting system in that it is a Java-system that has rectified a number of control deficiencies in classic Java RMI.

Firstly Voyager supports asynchronous method invocation, which can be seen as more of message-sending construct than an object-oriented one.

Object Voyager also supports futures, in order to handle object invocation return values in asynchronous method invocations. Futures are data structures that are passed to asynchronous method invocation call to hold the return values (and/or exceptions) when they become available. Threads access these futures by a another new special construct and can be made to suspend until the return value eventually becomes known. This is, of course, explicit data-flow.

In a limited sense, Object Voyager is just as much of a multi-paradigm programming system as Mozart (although the group behind Voyager would never present it in this way). Completely new types of entities and operations are introduced into the language. Voyager programs may look like Java ones but can easily have completely different behavior. Presumably Voyager programs only very occasionally make use of the new possibilities.

The Voyager support for all three sharing paradigms for distribution is not only ad-hoc but also limited. Data-flow is explicit only, the message-sending constructs are limited to one-to-one and many-to-one communication, and the only kind of distributed objects supported are stationary objects.

16.11 Partial Failure

16.11.1 Introduction

To evaluate integrated programming systems with regards to partial failure can only be done in comparison with the assembler of distributed programming: local computation and message sending. The most important question that must be answered is if integration hinders dealing with partial failure as compared to working at the lower level. If integration makes dealing with partial failure more difficult, then integration implies a *con* which needs to be weighed against all the *pros* that we have seen so far.

Of course, it would be even better if integration is actually beneficial with respect to partial failure - if high-level integration makes dealing with partial failure easier than at the lower level. If so, we would have an additional *pro* to add to the list of advantages to the integrated distributed programming system approach.

What does dealing with partial failure entail? First failure must be detected. Upon detection the process may initiate a cleaning up process - stopping activity that no longer serves any useful purpose (e.g. calculating the answer to a query posed by a failed process). Finally, and optionally, the process may initiate an error-recovery process to achieve fault-tolerance (e.g. having sent a query to failed server process attempt to send the same query to an alternative server).

16.11.2 Failure Detection

Without losing the essential qualities of failure at a low level we can make use of the concept of a failure detector. Processes can send messages to other processes that it knows about and by the action of failure detectors will be informed when these processes it knows about fail. Failure detectors in most distributed programming systems (and Mozart) are of the simplest kind - basing its decision exclusively on information gleaned from monitoring the communication channel. More sophisticated failure detectors are possible but we won't consider this further. Failure detection is often built into the messaging service (e.g. TCP/IP).

There are two kinds of failure conditions that a failure detector can report. The first, is process failure. This condition indicates that it is known for sure that the process has crashed. The second, is network failure, indicating that communication with the process is currently disrupted. The first condition is called *perm-fail* in the Mozart system, and the second *temp-fail*. Note that the second failure condition, unlike the first, in general, may, but need not, be a temporary phenomenon. When a process, from the point-of-view of another process, is in the state of network failure, we can

actually deduce very little with certainty. Network failure may hide process failure. In some deployment environments (e.g. LANs) network failures as described here may be rare, guaranteed to be very transient, or even non-existent. However, on WANs network failures do occur.

The following invariants hold:

- If process A from the viewpoint of another process has been determined to be in the state of perm-fail then eventually all other processes will be able to determine that process A is in a non-normal state, i.e. either perm-fail or temp-fail.

- If process A has from the viewpoint of another process been determined to be in the state of perm-fail then it will remain in that state forever.

Note that we do not assume the existence of any kind of consensus algorithm working here. It is possible for a process to be discovered to be in temp-fail state from the viewpoint of one process without this fact becoming known to any others.

In low-level message-sending systems failures may be reported asynchronously or alternatively only when an attempt to send a message is made. Asychronous failure detection requires that the messaging service continually monitors all the known processes.

16.11.3 Failure Detection in Integrated Programming Systems

In integrated programming systems, failure is detected on the level of language entities and not on the level of the status of remote processes. This may not be so apparent in message-oriented systems (like Erlang) or even in distributed object systems that only offer stationary objects, as the level of abstraction over the messaging infrastructure is low.

In message-oriented Erlang, failure is detected on the level of the fine-grained Erlang processes. Typically there are many Erlang processes within one OS-process. In RMI failure is detected on the level of failed objects (i.e. objects whose invocation gives rise to new kinds of exceptions). Note that process identifiers and remote object references can be freely passed between machines, and that the identity of the machine that hosts the failed remote object or failed Erlang process is hidden.

A failure model that detects permanently-failed and temporarily-failed language entities closely mirrors failure on the lower level but is more abstract. What we have abstracted out is the necessity of being aware of the mapping between process failures and the programming constructs. A site may hold two process identifiers (Erlang) or two ports (the Mozart message-sending construct). It may or may not be the case that the two processes are hosted on the same machine. It may or may not be the case that the two are inexorably joined and live and die together. Dealing with partial failure on the entities can be done independent of this - each entity is dealt with separately. Both may fail because they are hosted on the same machine - but both may also fail for other reasons.

In previous sections, we have discussed many different types of potentially shared semantic language entities, objects, ports, classes, procedures, records, single-assignment variables etc. Many of them come in a number of varieties that are semantically equivalent but have different distribution behavior. Stateless language entities can be associated with the lazy, eager and immediate protocols. Objects could be migratory, stationary, lazily invalidated, or eagerly invalidated. All entities except for the immediate and eager stateless entities can potentially fail.

We can use the following criteria over desired properties to evaluate our failure model for integrated programming systems.

- Minimize the number new failure states.
- Non-failed entities should always work.
- Failure not more likely to occur.
- Failure not more unpredictable.

The two failure states, which we call perm-fail and temp-fail, are inherent in distributed systems. We may have to introduce additional ones, but clearly would prefer not to.

The failure model should provide the assurance that entities that are not failed work normally, i.e. transparency of language semantics. It should be possible to determine that an entity has failed.

The system must be reasonably robust. It should be as robust as possible without going so far as to try to make use of complex fault-tolerant mechanisms (which is beyond the scope of what should be done on the language/system level, see also section 2.4). Put another way, it should not be easy to achieve greater robustness by moving down a level of abstraction.

Furthermore, by the awareness criteria, failure should not be more unpredictable. In particular we do not want strange third party dependencies that make reasoning about failure difficult.

Most of the consistency protocols of Mozart are fairly simple and are mapped straightforwardly to the underlying messaging framework. Their failure model is simple, and fulfill all the criteria above. Entities are always in one of three possible states, normal, temp-fail or perm-fail. Perm-fail is a permanent condition. Entities may oscillate back and forth between normal and temp-fail. Entities of this type are stateless entities annotated to be lazy, stationary objects and ports. Migratory objects and single-assignment variables are analyzed later. First we provide an example of a distributed programming system with very poor partial failure properties.

16.11.4 An example of poor integration w.r.t. partial failure

In the Emerald system [72, 71] distributed objects could not only be invoked remotely, but they could also be moved. This, in a more explicit way, brings many of the benefits

of the Mozart migratory object protocol. Repeated invocations by a threads on one site can be done locally.

The implementation supported this by using forwarding links. In general, these forwarding links form a chain that could be long (i.e. bounded only by the number of sites that reference the object). This made for poor awareness and control if the object is repeatedly moved. The drawback was that if one of the sites along the forwarding path fails the object becomes unreachable (at least, by the normal mechanism, as broadcast search could relocate the object in closed LAN-based distributed systems).

Returning to our three questions, the Emerald solution is characterized by difficulties on all four points (ignoring broadcast backup). First, there is an additional failure state, namely a partitioned failure state. Some of the processes see the object as normal, others as perm-failed. This can occur when a forwarding chain is broken in the middle. Second, an entity may be determined as perm-fail by some processes and normal by others - and this may last forever. Third, the existence of long forwarding chains makes entities extremely vulnerable to failure. Fourth, failure becomes very unpredictable. Failure is now dependent on an object's history as reflected in the structure of the forwarding chain.

16.11.5 Migratory objects

The migratory object protocol of Mozart presented in paper 2 (chapter 7) has poor partial failure properties. Paper 4 (chapter 9) shows an augmentation of the protocol with reasonable partial failure properties.

In particular, in our first naive migratory object protocol of paper 2, it was possible for the protocol to be in such a state that entity would appear to be in the normal state even thought the state had been lost and the entity irrevocably broken. This was fixed in the protocol as described in paper 4. (The Mozart system implements the improved protocol).

Despite this fix, there remain aspects of the augmented migratory object protocol together with the language interface that bear examination, together with the way failure is reflected up to the programmer

First, unlike the simpler protocols (e.g. stationary objects) there are three parties involved in the protocol. First there is the current holder of the state. There is only one such holder at any one time (the protocol manages a handover when the state moves from one site to another so we can, for this purpose, ignore the transient state during state movement). Second, there is the coordinator. In the naive version, the coordinator keeps track of the one and only current holder site of the state, but in the augmented protocol it may upon rapid movement of the state need to keep a record of a chain of sites. Note that if the state does not move very frequently - the normal case - that the chain in the augmented protocol is occasionally two sites long but normally one. The coordinator knows that the state is being held by one of the sites in the chain but not necessarily which one. The chain is the key to preserving the property of the naive protocol that requests for the state can be handled in a pipelined fashion to reduce latency without losing the ability to detect failure. Finally, in addition to

the two aforementioned sites (the coordinator and state-holder) we have an arbitrary number of other sites that contain references to the object.

In the Mozart system the coordination site is fixed - it is always the site where the entity was originally created. If the OS-process holding the state crashes then the state is lost and the object enters the perm-fail state. This information will be propagated to some or all of the sites that hold references to the object. Sites that have placed watchers (eager failure detection) on the object will be informed and sites that attempt to do operations will be informed. Note that sites that only reference the object, have no watchers on the entity installed, and never attempt to use the entity will not be notified. This is as it should be; this clearly reflects the the programmer's intent - despite the reference the path of execution that has been taken does not require the object. It also crucial to scalability - otherwise the coordinator would have to maintain a list of all sites that reference the object, rather than just the site that is currently holding the state and a few others. A property of the protocol in paper 4 is that when the coordinator observes a crash in a member of the current chain (except of course, for the trivial but common case that the chain consists of a single site) it must invoke a mini-protocol to determine if the state is truly lost or not.

If a site that is neither holding the state nor taking the role of the coordinator crashes, this has, as expected, no affect on the failure state of the object.

The third and last possibility is that the site holding the coordinator crashes. This makes the state unavailable to all but (possibly) one site - the current state-holder. The entity must enter the state of perm-fail from the viewpoint of all but the state-holder. Clearly the entity will not work for them, and indeed, with the coordinator dead they cannot know if the state is lost as well. What about the site that is holding the state? There are two possibilities, both with drawbacks. Either the entity is declared perm-fail, despite the availability of the state on the site, or it is not, which is a split-brain condition. We choose an intermediate position - and introduced a new failure state, or more precisely, a new substate of the perm-fail state. The reason for the introduction of the new state is to allow for the programming of fault-tolerance. As the state is available at the state-holder site it can be extracted as a prelude to error-recovery. (Note that the state is still intact so there is no problem with lost updates as in some replication schemes). Similarly, there is also a new substate to the temp-fail failure state (when the coordinator is subject to network failure)

We now consider migratory objects (based on the improved protocol of paper 4) from the viewpoint of our four criteria. Depending on your viewpoint we either introduce no new failure states or introduce two. However, the new states or substates should be seen more as a hook into the implementation for building fault-tolerant systems. There are still three basic failure states. We fulfill the criteria that non-failed entities always work. Failure is of migratory objects more likely compared to distributed objects realized by stationary objects (or basic message-passing). There are now two process failures that fail the entity rather than one. This seems both acceptable and to be the best that can be done to obtain the benefits of locality without sophisticated fault-tolerance instrumentation. Note, however that the protocol state of the coordinator is small so possibly schemes for building fault-tolerant coordination functions

might be profitably used here (see also future work in chapter 17).

Failure is fairly predictable - crashes of the original creation site and other users of the object can cause entity failure. We considered, but did not implement in the system, an addition to the protocol, whereby state that is not used for a long time is shipped back to the coordinator, in order to completely remove the dependence on the object's history. It is one thing to depend on other sites that are using the object at approximately the same time, it is quite another, if the object is long-lived and sporadically used, to be dependent on sites that worked with object a long time ago.

Two small further improvements in the protocol that are not covered in paper 4 were implemented. Triggers are introduced that cause the current state-holder to send the state back to the coordinator without the coordinator explicitly asking for it. Normally, the coordinating site will only ask for the state upon operations on the state just like any other site. The first trigger was, for obvious reasons, controlled shutdown. The second trigger was when the local garbage collector determined that the entity was no longer referenced on that site. The last was important not only to relieve the site of the burden of holding the state of an object it will never need but also because the coordinator and state-holder would otherwise form a cycle in the distributed garbage collection sense. Without this, the entity might not be reclaimed, despite the lack of references in the system as a whole. (Note that the cycle here is an artifact of the implementation and not a cycle due to cyclic references in dead data structures).

16.11.6 The Variable Protocol

The (data-flow, logical or single-assignment) variable protocol is described in detail in paper 3 (chapter 8). This protocol also makes use of a coordinator. However, unlike the migratory object coordinator, the coordinator eventually knows all sites that reference it. More precisely, the coordinator has a list of some of the sites that reference it. Other references, including those that exist in protocol messages, will by the action of the protocol later be registered via a dedicated message with the coordinator. (Deregistering may also take place via the action of local garbage collection).

It is important to realize that the semantics of the logical variable is that all references have equal rights and might potentially bind it and the variable can only be bound once. Note, that the a variable to variable binding, which effect merges the future value of both variables, is implemented as binding one variable to the other. The most common pattern for variable usage is where there is one reader thread (possible waiting for the variable to be bound) and one writer thread. However, situations where there are many reader threads occur fairly regularly as well. In the distributed system this means that the most common situation is where the variable is referenced from two sites only, the coordinator and one other site, but that scenarios with many site references are not uncommon.

We now consider the failure properties of variables. First we consider the number of failure states. We have two new failure states. They should be seen as substates of the normal failure perm/temp failure states. They were introduced to make maximal use of the fact that the coordinator knows all its references. These new failure states

indicate that one or more of the processes that reference the variable is in the state of perm-fail or temp-fail. The coordinator is still alive so the entity does still work.

The new failure states might, but need not, indicate that threads are in danger of suspending forever. If the failed site contained the one writer thread then threads on other sites will wait forever. The failure state is a hint that may or may not be heeded. Clearly, the failed site could also be a reader thread.

Note that the correct default is to consider these new failure states as new versions of the normal states. Note also that the new failure state is attempting to capture something that few distributed object systems attempt to do with objects. Threads may be programmed to wait upon something happening to an object (e.g. RMI object) just as they may be programmed to wait upon instantiation of a variable. Such threads also run the risk of waiting forever.

The criteria that non-failed entities work is fulfilled. This is in the sense that live processes can continue to work with the non-failed variable - e.g. one site can bind and other sites read. The entity fails only upon coordinator failure, i.e. is not more likely to occur than the base case. Failure is predictable; coordination always take place at the site where the variable was first created.

16.11.7 Asynchronous and synchronous failure in integrated systems

There is a distinction between asynchronous and synchronous failure detection on failed entities, just as is the case with detection on failed processes in low-level systems. Synchronous failure detection reflects failure that is detected upon attempting to use or operate on a shared language entity. Asynchronous failure detection reflects failure on entities irrespective of use.

This brings us to an important control consideration. A common pattern in all sharing models is a cycle of preparatory local computation followed by an operation on a shared language entity making use of the result of the local computation. The preparatory local computation may be large, so it is useful to be able to stop this early if the operation will in any case fail.

For this reason Mozart has two failure detection mechanisms, *watchers* which report failure eagerly and *handlers* which report failure upon entity use (for instance, via exceptions). A watcher is a user-defined procedure that is associated with a language entity. Upon entity failure (or more precisely, upon the types of failure that the watcher is configured to look for) the watcher procedure is invoked in its own thread. Watcher procedures may be coded to engage in cleanup activities. For instance, the watcher may kill or inject exceptions into running threads via first class thread references. The watcher may also initiate appropriate mechanisms for achieving fault-tolerance.

Although the failure model is on the level of language entities, the implementation still relies on failure detection on the level of OS-processes. When process failure is discovered the implementation maps this failure onto entities, possibly changing the failure state of the entity (normal, perm-fail or temp-fail). Watchers will be invoked immediately while other failure states will be discovered first when operations are

attempted on the entity. The failure model introduces no overhead as compared to low-level failure monitoring. except for the small overhead involved in performing the process to entity mapping, which is only needed when failure is actually detected.

Many distributed programming systems lack asynchronous error detection. For instance, in RMI, failure is reflected solely in exceptions on remote method invocation.

16.11.8 Other failure considerations and conclusion

The proposed partial failure model for integrated distributed programming systems reflects partial failure on a per-entity basis. This integrates partial failure into the language model and thus conforms to the ideals of integrated distributed programming systems.

However dealing with partial failure is composed of three separate activities. First is failure detection on the language level. This is what we have concentrated on. The model can undoubtedly be refined but we believe we have demonstrated the feasibility of this approach even for systems that support a wide range of both entity types and consistency protocols.

The second activity is the cleanup activity, where work that is no longer useful is stopped after failure has been detected. Does Mozart and/or other distributed programming systems have good support for this? This is an open question. In Oz the programmer can group threads and via first class thread references control such groups (i.e. kill them) so there is some support, at least in theory. In the Mozart Programming System threads were not fully instrumented for distribution, i.e. the Thread library was not made network-transparent (in the Mozart release). The issue here was not conceptual but rather lack of resources. Possibly, new grouping support (of threads and entities) is needed to simplify this aspect of failure handling.

The third activity is the establishment of fault-tolerance. An error-recovery procedure is initiated that can replace the activity that was stopped by process or entity failure. How closely the recovered system needs to mirror what would have happened had no process failure occurred is application dependent.

There are many algorithms and approaches to making applications fault-tolerant. Much of this is beyond the scope of the distributed programming language/system per se. From the language and system point-of-view we do however need to analyze if the distributed programming system has all the necessary hooks in order to plug-in mechanisms from the field of fault-tolerant systems and methods. We need to consider how to design the system such as to be able to easily integrate techniques of fault-tolerance into the system. This issue is beyond the scope of this thesis but is part of future work (see section 17).

16.12 Three Sharing Models

16.12.1 Introduction

The three sharing models were discussed in the context of centralized concurrent programming systems (chapters 12 and 13). There we saw that, although any one sharing model was programmable in terms of another, this involved performance penalties. The really serious performance penalties, involving algorithmic complexity were programming objects in pure data-flow or message-passing systems and programming message-oriented in pure data-flow languages. In the concurrent setting the other three abstractions had only a slight performance penalty (in reasonable implementations). These three were programming data-flow with objects, message-sending with objects, and data-flow with message-sending.

As pure data-flow languages are today very rare, for the aforementioned reason, we don't feel the need to reconsider making do with a pure data-flow language but take for granted that data-flow is always complemented with at least one other sharing paradigm.

We will however look at the other four cases in the context of distribution.

16.12.2 Protocol properties

We need to digress and explain some important properties of both the variable protocol (described fully in paper 3, chapter 8) and the state invalidation protocol (briefly described earlier in section 16.6.4 but never implemented in the Mozart system). The implementation of these protocols depend on being informed of first-time import of the entity reference. They also depend on garbage-collection, i.e. they need to be informed when the entity is no longer referenced on the current site. Note that if a reference has been imported, then garbage-collected, and then reimported this is also considered a first-time import. In the variable protocol the first-time import (but not subsequent imports) triggers a register message to the coordinator and entity reclamation during garbage-collection will trigger a deregister message.

In general, the more complex protocols depend on both export/import awareness as well as garbage-collection awareness. This means that a fundamental property of these transparency-preserving protocols is that they themselves are fundamentally non-transparent as transparency means that the programmer does not see, nor care about, the process or machine boundaries.

16.12.3 Objects and message-sending

We showed earlier that programming shared *stationary* objects in Erlang (i.e. message-oriented system) is straightforward. The distribution behavior is identical (one message to the object/process and one message back). Furthermore whatever slight performance penalty there might be in the centralized case is surely insignificant in the distributed

case. The Erlang people are quite right to say that the message-oriented distribution support in Erlang subsumes RMI but is strictly more powerful.

However distributed objects need not be stationary, indeed, for many application patterns mobile or invalidation-based objects are better - and this is on the level of algorithmic complexity. These kinds of distributed objects are not programmable in Erlang proper, for the reasons described in the last subsection. Possibly, with the aid of some non-transparent low-level hooks into the implementation, they could be but in that case the end result would in any case be a re-implementation of the consistency protocols, partly in Erlang and partly on a lower-level (i.e. the hooks).

In chapter 13 we saw that message-sending is programmable in terms of objects. In a distributed context most of the argumentation carries over to programming distributed message-sending in terms of stationary objects. However it is crucial that the abstraction is truly asynchronous and this requires the creation of an auxiliary thread. This may be acceptable performance-wise if threads are lightweight (Voyager could not take this approach as threads are not lightweight in Java).

16.12.4 Data-flow abstractions

In chapter 13 we discussed abstractions to implement explicit data-flow in object-oriented systems. If the distributed counterpart makes use of stationary objects the resultant protocol corresponds roughly to a lazy version of the variable protocol (see paper 3, chapter 8) which is not what you want. Single-assignment variables are generally very transient entities and the dependency between sites should be broken as quickly as possible (but this does not mean that variables cannot or should not become bound to lazy stateless data structures). The property of the variable protocol that the distribution infrastructure is quickly dismantled upon variable binding is also lost. Variable-variable bindings (i.e. when two unbound single-assignment variables are merged) are also problematic.

To base data-flow on migratory objects is even worse, and will involve many additional network hops. To base data-flow on eager invalidation objects most closely captures the qualities of the variable protocol. Here the main difference is in the automatic dismantling of the distribution infrastructure. This reasoning is, of course, somewhat academic. We know of no system that has smoothly incorporated invalidation protocols in a system supporting distributed objects - let alone one that has demonstrated explicit data-flow on top. Recent work in Java [97] does, however, supply some invalidation-based protocol support for distributed objects.

Programming data-flow in message-oriented systems like Erlang is just as problematic as programming invalidation-based objects in Erlang using message-sending constructs (as described in previous subsection).

Chapter 17

Conclusion and Future Work

We believe that, in the course of time, mature distributed programming systems will be developed that are characterized by good abstraction, awareness and control. This will probably take some time - we are still in the beginning of this development. Along the way various systems and languages will be developed only to later be superseded by better systems.

We also believe that we have, with the Mozart Programming System, contributed to this search for the ultimate (or just very good) distributed programming system. We believe that there are a number of system and language properties that Mozart exhibits that good distributed programming systems must and ultimately will have.

In this final chapter, the first part is devoted to summarizing those properties that Mozart either has or has to some degree, that are essential in good distributed programming systems. These are properties that eventually, as the field matures and the dust finally settles, that all distributed programming systems will need to survive the coming Darwinian struggle for supremacy. These are properties we feel that we have demonstrated in the Mozart work.

The second part of this chapter is devoted to future work. We begin by describing extensions that could be incorporated into Mozart. Some of these features we just did not have the resources to realize, others have been gleaned from our experiences in the Mozart project. We also mention some implementations aspects that would benefit from reengineering.

Finally, we briefly describe aspects of distributed programming that are still open research questions. These are aspects that we expect the community to tackle in the coming years, but where, today, it is difficult to see the correct approach let alone the final solution.

17.1 Necessary Qualities of Distributed Programming Systems

We believe that good distributed programming systems will have the following characteristics:

- Good concurrent core
- Higher order (all entities are first class values)
- Complete range of semantic entities
- Complete protocol choice for good control

The concurrent core should support fine-grained concurrency and offer a fair preemption-based built-in scheduler. The programmer should never need to program his/her own thread pool nor a scheduler. For good performance on the same machine the system should support threads. Possibly, depending on future garbage-collection algorithmic development, the system should also support fine-grained processes as well.

The programming model for dealing with code and data should be as integrated as possible, i.e. the system should support first-class procedures/functions/classes. The distribution of code and stateless data are not inherently different. Sometimes the programmer wants to create a procedure or class in one thread (machine) and share it with other threads (machines); sometimes he/she wants to share a stateless data structure. The programmer wants uniqueness guaranteed. In other cases, the programmer wants to work within a namespace - a named procedure/class or named data structure. From the control point of view the system should leverage that the recipient may already have a copy.

The system should offer a complete range of semantic entities. This may be advantageous from the point-of-view of program readability - as for instance, when the distinction between single-assignment and multiple-assignment variables is visible. More important is that awareness model will rank the various entities. The golden control rule is to prefer the one associated with the weakest consistency model that captures the programmer's intent. This means that there will be many varieties of shared state.

The system should offer a complete protocol choice for languages entities. One way is to associate with each type of semantic language entity a rich set of annotations. These annotations specify the consistency protocol associated with the entity, or rather the flavor as the consistency protocol is partly determined by entity type. The golden rule here is to determine, possibly with the aid of some tool in a late development state, the usage pattern and then pick the most optimal consistency protocol. Note that Mozart is not complete in this sense. Even for sequentially consistent state Mozart provides only two consistency protocols (we recognize the need for at least four).

Protocol choice, as discussed extensively in the last chapter, is primarily concerned with performance in the distributed system sense, i.e. the number of hops, delays, the number of messages and sizes of messages. This choice controls which of a number of functionally equivalent protocols should be chosen to get the best performance - which is a non-functional property. There may be other such choices covering other non-functional properties such as failure and security (see future work).

The completeness of entity types and protocol choice are related. There are entity types that may be derived from other entity types (e.g. sets from lists). Useful entity types need not necessarily be primitive in the system but may be encaptured in libraries.

From the distribution point-of-view, however, each such library entity must face an acid test. Does the composite consistency protocol of the composite entity as defined in the library provide the necessary control? Could a better consistency protocol be derived by making the entity primitive in the implementation so as to be able to directly couple the best consistency protocol with the entity? If the library versions do not provide a complete choice then the entity must be made primitive. This does not necessarily mean that this type of language entity has the same dignity as others, functionally it can be derived using other primitive entities. It is only primitive in the implementation to provide the proper control.

In the Mozart programming system, we went further than other systems in providing for a complete range of entities and a complete range of protocol choice. Though Mozart is not complete with respect to either, it does show it it both possible and practical to make distributed programming systems that come much closer to completeness. We also believe that with the right system software architecture completeness is possible without undue complexity. This is discussed more in future work.

We should also note that there are many language features of Oz/Mozart that are outside the scope of this discussion. This does not mean that author does not have an opinion (just like almost everyone else who works with programming languages/systems). We do not, in this thesis, consider those aspects of programming languages/systems where distribution does not change the balance between the pros and cons of different paradigms or constructs. Neither do we make strong claims where distribution does affect the balance, but it is not unclear how to weigh the new pros and cons. The best example of this is the tradeoff between explicit and implicit data-flow. Distribution makes for more unpredictability in timings. This favors implicit data-flow which is resilient to production order changes. However, implicit data-flow also complicate failure handling, introducing as it does transient dependencies between sites. Distribution adds both new pros and new cons to this discussion.

17.2 Future Work

In the last chapter we saw that Mozart also falls short with respect to our vision of well-designed integrated programming systems. That Mozart falls short is, of course, not solely due to lack of time and resources, but also, that our vision today is not quite what it was when we began the Mozart work. We, too, were overly influenced by the centralized programming heritage.

Today we have the benefit of hindsight and can formulate the design principles and target criteria for integrated distributed programming languages/systems more clearly and concisely. This, we believe, is one of the major outcomes of the Mozart work.

One important aspect of future work is to complete the system according to the principles outlined in this thesis. First, there is the matter of those language entities that were never given a distribution behavior, that were never made transparent. Examples are the built-in data types arrays and dictionaries. Today these entities do not work when shared between sites. One of the reasons that these were left for later was, that

though they were clearly stateful entities, there were uncertainties about the granularity of the protocol support. An array, for example, can be seen as a stateless data structure containing a vector of stateful attributes, one for each array element. In this view the consistency of each array element is dealt with separately. This has the advantage that sites that are updating/accessing different array elements do not interfere with each other at all. This fine-grained view has major disadvantages as well. We have additional space requirements for all the required distribution infrastructure for the array elements. More latency is introduced for operations involving more than one array element. Alternatively, an array can be seen as a single monolithic stateful entity. Intermediate views are also possible. From the control perspective we need to offer all useful types.

Another aspect to completing the system is to implement a complete set of protocols, and provide the user some means to (orthogonally to functionality) specify the most optimal protocol. Invalidation protocols for stateful entities were, for example, never implemented.

The vision of completeness as regards types of language entities and the associated protocol choices is a vision that arose rather late in the Mozart work. In the early ground-laying work, the vision that we worked with was more to extend all the language entities of Oz to transparent distribution. The software architecture of the Mozart system, as released in 2000-2001, reflects this. The way the system was designed makes it a major undertaking to add new language entities and new protocols to the system. We realized that to make the goal of completeness realizable that the software architecture must be reworked to allow for incremental and encapsulated entity and protocol addition. The system needs to be open and easy to extend with new entities and new protocols. This incrementality will not only significantly ease system development but will allow easy addition of new consistency protocols as new distributed algorithms upon which to base them are developed by the community.

A language and system aspect that needs to be looked into is dynamic relinking for dealing with resources. Functors do provide a means to transfer code between sites with dynamic linking, but this is not enough. The problem can be formulated as an identity problem. The mechanisms in Mozart are mostly based on the most conservative identity model. Entities are only considered identical if they have the same source. For many entity types this is the key to transparency. For example, if the reference to a local object is exported to another site, then we have a shared object. The consistency of this object is maintained by a consistency protocol. For this to work it must be considered the same object. A site is not allowed to substitute this reference with something else. The question is: can we relax this very strict view on the identity relation?

Functors in Mozart do provide a means to relax the identity relation. It will link properly packaged code based on the names of functors, substituting in local copies with the same name. However once linked code cannot be unlinked. This is inconvenient. For instance, it may limit delegation; a site cannot, in general, take a functor, link it, begin to use it and then later try to delegate execution of some subtask (e.g. send a first-class procedure to a computation server). This is a limitation of Mozart

that should be rectified.

Relaxed identity, is not only important for dealing with code but also for dealing with resources. Typical resources are the file system and the window system (graphics). The distinction between resources and code is slightly blurred in Mozart, as resources are typically packaged inside a functor. Upon analyzing resource usage it quickly becomes apparent that upon export that the semantics that you want is application dependent, even for the same resource. Consider, for example, the file system. In one application a specific file needs to be opened and read. If this work is to be delegated to a machine that does not have access to the same file system, then the reference must refer to the same file system. Here we can imagine treating the file system in the same way as we do stationary objects. Upon export a distribution infrastructure is initialized to transfer request back to the original site. In another application a temporary file is created, used, and then later expunged. In this case it does not matter which file system is used. Substituting the local file system is the correct thing to do.

Here we see a fundamental uncertainty about the semantics of the 'language entity' the file system. For this reason, in the Mozart system, resources like the file system simply do not work outside the original site (i.e. cause exceptions when used outside the original site). Care was taken that such references could be exported, so they did not break the system when not used. Care was also taken that such references when reimported to the original site would work again. The rationale behind this choice was to expose the unclarity. Inadvertent export and subsequent use of resources would be discovered in exceptions. If dynamic linking was desired the programmer could use functors. If static linking is desired, the programmer can wrap the resource inside a stationary object. Nevertheless these work-arounds are limited and inconvenient and better integration of resources, in particular, and relaxed identity, in general, should be pursued. Interesting is that when we take the language view on the entity 'file systems' that they can be both stateful (reading a specific file) and stateless (creating a new file). Only in the stateless case is relinking at all possible.

Finally, there is the question of security. This needs to be considered both in the distributed programming language and the system or implementation, taking into the account the possibility of malicious action. The language Oz/Mozart has many of the properties that are known to be desirable for security. This is covered in paper 1 (chapter 6) in the section on language security. However, security was never a major focus in the Mozart work and strengths of the core language do not extend to system libraries. This needs to be looked into. As regards the implementation very little was done. There are at least three obvious actions that should be taken, but there may be more. These are (1) the unmarshaler must be made robust (so that malicious sites cannot cause a crash), (2) entities must be made unguessable in the distributed implementation and (3) communication must be encrypted. For control, a trust model is probably needed to be able to turn off security enhancements that induce large overheads upon establishing trust between parties.

17.3 Outstanding Research Questions

We said earlier that future distributed programming languages/systems will offer a complete range of semantic entities. This clearly includes stateless, single-assignment and multiple-assignment stateful. But what else does it include? What makes the range complete?

There are many different kinds of entities with distinct associated consistency semantics that are weaker than sequential consistency. Many of these may turn out to be useful over a range of applications so a distributed programming system should cater for them. For example, we have time-bounded eventual consistency, where readers are only guaranteed that the value that they see is not too much out of date. Clearly this type of consistency requires, in general, less synchronization and fewer messages than, say, invalidation-based sequential consistency protocols. Exactly what constitutes the full range of useful primitive entities and protocols is an open question.

There are still open questions concerning partial failure. To begin with, there is the issue of how best to reflect this on the programming level. An important consideration is granularity. Is failure on the level of individual entities practical or do we need ways to group entities/threads to capture partial failure on a coarse-grained level. Often the programmer would like to make the application at least partly, fault-tolerant. There are techniques for achieving fault-tolerance. We know that fault-tolerant abstractions can be created: but what kind of system support do they need? How can the programming system best interface with mechanisms for fault-tolerance?

Bibliography

[1] Distribution subsystem. `http://dss.sics.se`.

[2] Java messaging service. Available at `http://java.sun.com/products/jms`.

[3] Object voyager. `http://objectspace.com/voyager`.

[4] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Mass, 1985.

[5] Mack W. Alford, Leslie Lamport, and Geoff P. Mullery. *Basic Concepts, in Distributed Systems–Methods and Tools for Specification, An Advanced Course*, chapter 2. Lecture Notes in Computer Science, vol. 190. Springer Verlag, 1985.

[6] Iliès Alouini. Orthogonal fault tolerance based on watchers. Unpublished, 1998.

[7] Iliès Alouini and Peter Van Roy. Le protocole réparti de Distributed Oz (in French). In *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP 99)*, pages 283–298, Nancy, France, April 1999.

[8] Edward G. Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall, 1994.

[9] James Andrews. The logical semantics of the Prolog cut. In *International Logic Programming Symposium (ILPS 95)*, December 1995.

[10] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: A single system image of a JVM on a cluster. In *International Conference on Parallel Processing*, pages 4–11, 1999.

[11] Joe Armstrong, Mike Williams, Claes Wikström, and Robert Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, N.J., 1996.

[12] Arvind and R. E. Thomas. I-Structures: An efficient data type for functional languages. Technical Report 210, MIT, Laboratory for Computer Science, 1980.

[13] Tomas Axling, Seif Haridi, and Lennart Fahlen. Concurrent constraint programming virtual reality applications. In the *2nd International Conference on Military Applications of Synthetic Environments and Virtual Reality* (*MASEVR 95*), Stockholm, Sweden, 1995. Defence Material Administration.

[14] Henri E. Bal, Frans E. Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[15] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[16] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-structures: Extending a parallel, nonstrict, functional language with state. In *Functional Programming and Computer Architecture*, Berlin, August 1991. Springer-Verlag.

[17] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[18] Per Brand, Nils Franzen, Erik Klintskog, and Seif Haridi. A platform for constructing virtual spaces. In *Virtual Worlds and Simulation Conference (VWSIM '98)*, January 1998.

[19] Per Brand, Peter Van Roy, Raphaël Collet, and Erik Klintskog. A fault-tolerant mobile-state protocol. In preparation, 1999.

[20] Brent Callaghan. WebNFS—The file system for the World-Wide Web. White paper, Sun Microsystems, Mountain View, Calif., May 1996.

[21] Luca Cardelli. A language with distributed scope. *ACM Transactions on Computer Systems*, 8(1):27–59, January 1995.

[22] Christer Carlsson and Olof Hagsand. DIVE—A platform for multi-user virtual environments. *Computers and Graphics*, 17(6), 1996.

[23] Nicholas Carriero and David Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, February 1992.

[24] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In the *13th ACM Symposium on Operating System Principles*, pages 152–164, New York, 1991. ACM.

[25] Randy Chow and Theodore Johnson. *Distributed Operating Systems and Algorithms*. Addison-Wesley, San Francisco, Calif., 1997.

[26] Alain Colmerauer. *Prolog and Infinite Trees*. Academic Press, 1982. In *Logic Programming*, Keith L. Clark and Sten-Åke Tärnlund, eds.

[27] Douglas E. Comer. *Internetworking with TCP/IP. Vol. 1: Principles, Protocols, and Architecture*. Prentice-Hall, Englewood Cliffs, N.J., 1995.

[28] The Mozart Consoritum. Various documentation. Technical report, 1999. In Mozart documentation, available at `http://www.mozart-oz.org/documentation`.

[29] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems Concepts and Design 2nd ed.* Addison-Wesley, Reading, Mass., 1994.

[30] Jon Crowcroft. *Open Distributed Systems*. University College London Press, London, U.K., 1996.

[31] M. Dahm. Doorastha: a step towards distribution transparency, 2000.

[32] Steve Deering. Host extensions for IP multicasting. Technical Report RFC1112, IETF, August 1989.

[33] DFKI Oz version 2.0, 1996. Available at `http://www.ps.uni-sb.de`.

[34] DFKI Oz version 2.0, February 1998. Available at `http://www.ps.uni-sb.de`.

[35] M. Diaz, B. Rubio, and J. M. Troya. DRL: A distributed real-time logic language. *Comput. Lang.*, 23(2–4):87–120, 1997.

[36] Ericsson. *Open Telecom Platform—User's Guide, Reference Manual, Installation Guide, OS Specific Parts*. Telefonaktiebolaget LM Ericsson, Stockholm, Sweden, 1996.

[37] J.C. Fabre and T. Perennou. A metaobject architecture for fault tolerant distributed systems: The Friends approach. Technical report, Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS), Toulouse, January 1997.

[38] K. E. Kerry Falkner, P. D. Coddington, and M. J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. Technical Report DHPC-072, 1999.

[39] K. Fischer, N. Kuhn, and J. P. Müller. Distributed, knowledge-based, reactive scheduling in the transportation domain. In the *10th IEEE Conference on Artificial Intelligence and Applications*, New York, March 1994. IEEE.

[40] K. Fischer, J. P. Muller, and M. Pischel. A model for cooperative transportation scheduling. In the *1st International Conference on Multiagent Systems* (*ICMAS 95*), pages 109–116, June 1995.

[41] François Fluckiger. *Understanding Networked Multimedia: Applications and Technology*. Prentice-Hall, 1995.

[42] Mike Foody. Let's talk (Special report building networked applications). *BYTE*, 22(4):99–102, April 1997.

[43] Ian Foster. Parallel implementation of Parlog. In *International Conference on Parallel Processing*, pages 9–16. IEEE Computer Society, 1988.

[44] Tetsuro Fujise, Takashi Chikayama, Kazuaki Rokusawa, and Akihiko Nakase. KLIC: A portable implementation of KL1. In *Fifth Generation Computing Systems (FGCS '94)*, pages 66–79, December 1994.

[45] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. Available at `http://www.javasoft.com`.

[46] James Gosling and Henry McGilton. The Java language environment. White paper, Sun Microsystems, Mountain View, Calif., May 1996.

[47] Donatien Grolaux. Editeur graphique réparti basé sur un modèle transactionnel (A distributed graphic editor based on a transactional model). Technical report, Université catholique de Louvain, June 1998. Mémoire de fin d'études.

[48] David Gudeman. Representing type information in dynamically typed languages. Technical Report TR93-27, University of Arizona, Department of Computer Science, September 1993.

[49] Robert H. Halstead, Jr. MultiLisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[50] Seif Haridi. *Logic Programming based on a Natural Deduction System*. PhD thesis, Royal Institute of Technology, Stockholm, 1981.

[51] Seif Haridi and Nils Franzén. Tutorial of Oz. Technical report, 1999. In Mozart documentation, available at `http://www.mozart-oz.org`.

[52] Seif Haridi and Dan Sahlin. *Efficient implementation of unification of cyclic structures*. Ellis Horwood Limited, 1984. In *Implementations of Prolog*, J. A. Campbell, ed.

[53] Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient logic variables for distributed computing. *ACM TOPLAS (to appear)*, 1999.

[54] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, May 1998.

[55] Seif Haridi, Peter Van Roy, and Gert Smolka. An overview of the design of Distributed Oz. In the *2nd International Symposium on Parallel Symbolic Computation (PASCO 97)*. ACM, July 1997.

[56] Seif Haridi, Peter Van Roy, and Gert Smolka. An overview of the design of Distributed Oz. In the *2nd International Symposium on Parallel Symbolic Computation* (*PASCO 97*), New York, July 1997. ACM.

[57] Bernhard Haumacher and Michael Philippsen. Exploiting object locality in JavaParty, a distributed computing environment for workstation clusters. In *CPC2001, 9th Workshop on Compilers for Parallel Computers*, pages 83–94, June 2001.

[58] Martin Henz. *Objects for Concurrent Constraint Programming*, volume 426 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, November 1997.

[59] Martin Henz. *Objects in Oz*. PhD thesis, Saarland University, Fachbereich Informatik, Saarbrücken, Germany, June 1997.

[60] Martin Henz. Objects in Oz. Doctoral dissertation, Saarland University, Saarbrücken, Germany, May 1997.

[61] Martin Henz, Stefan Lauer, and Detlev Zimmermann. COMPOzE—Intention-based music composition through constraint programming. In the *International Conference on Tools with Artificial Intelligence*, New York, November 1996. IEEE.

[62] Martin Henz, Gert Smolka, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In Pascal Van Hentenryck and Vijay Saraswat, editors, *Principles and Practice of Constraint Programming*, pages 29–48, Cambridge, Mass., 1995. The MIT Press.

[63] Martin Henz and Jörg Würtz. Using Oz for college timetabling. In E. K. Burke and P. Ross, editors, the *International Conference on the Practice and Theory of Automated Timetabling*, volume 1153 of *Lecture Notes in Computer Science*, pages 162–177, Berlin, 1996. Springer-Verlag.

[64] Robert A. Iannucci. *Parallel Machines: Parallel Machine Languages. The Emergence of Hybrid Dataflow Computer Architectures*. Kluwer, Dordrecht, the Netherlands, 1990.

[65] N. Ichiyoshi, T. Miyazaki, and K. Taki. A distributed implementation of Flat GHC on the Multi-PSI. In *Fourth International Conference on Logic Programming*, pages 257–275. The MIT Press, May 1987.

[66] Institute for New Generation Computer Technology, editor. *Fifth Generation Computer Systems 1992*, volume 1,2. Ohmsha Ltd. and IOS Press, 1992. ISBN 4-274-07724-1.

[67] Joxan Jaffar and Michael Maher. Constraint logic programming: A survey. *J. Log. Prog.*, 19/20:503–581, May/July 1994.

[68] Pankaj Jalote. *Fault Tolerance in Distributed Systems*. PTR Prentice-Hall, 1994.

[69] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In *International Symposium on Logic Programming*, pages 167–183, October 1991.

[70] Sverker Janson, Johan Montelius, and Seif Haridi. Ports for objects in concurrent logic programs. In *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, 1993.

[71] Eric Jul. *Object Mobility in a Distributed Object-Oriented System*. PhD thesis, Univ. of Washington, Seattle, Wash., 1988.

[72] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[73] Setrag Khoshafian and Marek Buckiewicz. *Introduction to Groupware, Workflow, and Workgroup Computing*. J. Wiley and Sons, 1995.

[74] Erik Klintskog, Anna Neiderud, Per Brand, and Seif Haridi. Fractional weighted reference counting. In *LNCS 2150*, 2001.

[75] Michael Knapik and Jay Johnson. *Developing Intelligent Agents for Distributed Systems*. McGraw-Hill, 1998.

[76] Evelina Lamma, Paola Mello, Cesare Stefanelli, and Pascal Van Hentenryck. Improving distributed unification through type analysis. In *Euro-Par '97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 1181–1190. Springer-Verlag, 1997.

[77] Butler W. Lampson. Reliable messages and connection establishment. In Sape Mullender, editor, *Distributed Systems*, pages 251–281, Reading, Mass., 1993. Addison-Wesley.

[78] J. C. Laprie. Dependability: A unifying concept for reliable computing and fault tolerance. In *7th International Conference on Distributed Computing Systems*, pages 129–146, September 1987.

[79] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.

[80] Lone Leth and Bent Thomsen. Some Facile chemistry. Technical Report ECRC-92-14, ECRC, Munich, Germany, May 1992.

[81] Ho-Fung Leung. *Distributed Constraint Logic Programming*, volume 41 of *Series in Computer Science*. World Scientific, Singapore, 1993.

[82] Ho-Fung Leung and Keith L. Clark. Constraint satisfaction in distributed concurrent logic programming. *J. Symbolic Computation*, 21:699–714, 1996.

[83] B. Liskov and L. Shrira. Linguistic support for efficient asynchronous procedure calls in distributed systems. In *In Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 260–267, june 1988.

[84] John Lloyd. *Foundations of Logic Programming, Second Edition*. Springer-Verlag, 1987.

[85] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, Calif., 1996.

[86] Michael Maher. Logic semantics for a class of committed-choice programs. In *Proceedings of the Fourth International Conference on Logic Programming (ICLP 87)*, pages 858–876, Melbourne, Australia, May 1987. The MIT Press.

[87] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[88] Martin Müller, Tobias Müller, and Peter Van Roy. Multiparadigm programming in Oz. In Donald Smith, Olivier Ridoux, and Peter Van Roy, editors, *Workshop on the Future of Logic Programming, International Logic Programming Symposium (ILPS 95)*, December 1995.

[89] General Magic. *Telescript Developer Resources*. General Magic.

[90] Sun Microsystems. *The Remote Method Invocation Specification*, 1997. Available at `http://www.javasoft.com`.

[91] Michael Mehl, Ralf Scheidhauer, and Christian Schulte. An abstract machine for Oz. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *Programming Languages, Implementations, Logics and Programs, Seventh International Symposium, PLILP'95*, volume 982 of *Lecture Notes in Computer Science*, pages 151–168, Utrecht, The Netherlands, September 1995. Springer Verlag.

[92] Michael Mehl, Christian Schulte, and Gert Smolka. Futures and by-need synchronization for Oz. Draft, Programming Systems Lab, Universitt des Saarlandes, May 1998.

[93] K. Mehlhorn and A. Tsakalidis. Data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science – Volume A: Algorithms and Complexity*, pages 301–341. Elsevier, The MIT Press, 1990.

[94] Mozart Consortium. The Mozart Programming System version 1.2.3, December 2001. Available at `http://www.mozart-oz.org/`.

[95] Lee Naish. *Negation and Control in Prolog*, volume 238 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[96] Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA: The Common Object Request Broker Architecture*. Prentice-Hall PTR, Upper Saddle River, N.J., 1996.

[97] Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[98] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Kinross, Scotland (UK), September 1995.

[99] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In the *International Workshop on Memory Management*, Lecture Notes in Computer Science, vol. 986, pages 211–249, Berlin, September 1995. Springer-Verlag.

[100] Andreas Podelski and Gert Smolka. Operational semantics of constraint logic programs with coroutining. In *International Conference on Logic Programming (ICLP 95)*, pages 449–463, 1995.

[101] K. Popov, V. Vlassov, P. Brand, and S. Haridi. An efficient marshaling framework for distributed systems. In Victor E. Malyshkin, editor, *Parallel Computing Technologies, 7^{th} International Conference (PaCT 2003)*, volume 2763 of *LNCS*, pages 324–331, Nizhni Novgorod, Russia, September 15–19 2003. springer. A revised version to appear in Future Generation Computer Systems, May 2005.

[102] Konstantin Popov, Vladimir Vlasov, Mahmoud Rafea, Fredrik Holmgren, and Seif Haridi. Parallel agent-based simulation on a cluster of workstations. In *Proceedings of EUROPAR'03*, pages 470–480. springer, August 2003. Extended version will also appear in Systems Analysis Modelling Simulation Journal, 2004.

[103] Konstantin Popov, Vladimir Vlasov, Mahmoud Rafea, Fredrik Holmgren, and Seif Haridi. Parallel agent-based simulation on a cluster of workstations. *Parallel Processing Letters*, 13(4):629–641, December 2003.

[104] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw Hill, 2003. ISBN 007-123265-6.

[105] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.

[106] Kazuaki Rokusawa, Akihiko Nakase, and Takashi Chikayama. Distributed memory implementation of KLIC. *New Generation Computing*, 14(3):261–280, 1996.

[107] Vijay Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL 90)*, pages 232–245, San Francisco, CA, USA, January 1990.

[108] Vijay A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, 1993.

[109] Richard D. Schlichting and Vicraj T. Thomas. Programming language support for writing fault-tolerant distributed software. *IEEE Transactions on Computers*, 44(2):203–212, February 1995.

[110] S. Schmeier and Schupeta Achim. PASHA II—Personal assistant for scheduling appointments. In the *1st International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology* (*PAAM 96*), Lancashire, United Kingdom, 1996. The Practical Application Company.

[111] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press.

[112] Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 519–533, Schloß Hagenberg, Austria, October 1997. Springer-Verlag.

[113] Christian Schulte and Gert Smolka. Finite domain constraint programming in Oz. A tutorial. Technical report, DFKI and Saarland University, December 1999. Available at `http://www.mozart-oz.org/`.

[114] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.

[115] Gert Smolka. *An Oz Primer*. Programming Systems Lab, Saarland University, Saarbrücken, Germany, 1995. Available at `http://www.ps.uni-sb.de`.

[116] Gert Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer Verlag, 1995.

[117] Gert Smolka. Concurrent constraint programming based on functional programming. In Chris Hankin, editor, *Programming Languages and Systems*, volume 1381 of *Lecture Notes in Computer Science*, pages 1–11, Lisbon, Portugal, 1998. Springer-Verlag.

[118] Gert Smolka, Christian Schulte, and Peter Van Roy. PERDIO—Persistent and distributed programming in Oz. BMBF project proposal. Available at `http://www.ps.uni-sb.de`, February 1995.

[119] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 1997.

[120] Sun Microsystems. *The Java Series*. Sun Microsystems, Mountain View, Calif., 1996. Available at `http://www.javasoft.com`.

[121] Andreas Sundström. Comparative study between Oz 3 and Java. Technical report, Uppsala University and Swedish Institute of Computer Science, July 1998.

[122] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Englewood Cliffs, N.J., 1995.

[123] Andrew Taylor. *High-Performance Prolog Implementation*. PhD thesis, Basser Department of Computer Science, University of Sydney, June 1991.

[124] Gerard Tel. *An Introduction to Distributed Algorithms*. Cambridge University Press, Cambridge, United Kingdom, 1994.

[125] Tommy Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, September 1997.

[126] Jeffrey D. Ullman. *Elements of ML Programming*. Prentice Hall, 1998.

[127] Peter Van Roy. 1983–1993: The wonder years of sequential Prolog implementation. *J. Log. Prog.*, 19/20:385–441, May/July 1994.

[128] Peter Van Roy, Per Brand, Seif Haridi, and Raphaël Collet. A lightweight reliable object migration protocol. In Henri E. Bal, Boumediene Belkhouche, and Luca Cardelli, editors, *Internet Programming Languages*, volume 1686 of *Lecture Notes in Computer Science*. Springer Verlag, October 1999.

[129] Peter van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004. ISBN 0-262-22069-5.

[130] Peter Van Roy, Seif Haridi, and Per Brand. Distributed programming in Mozart – A tutorial introduction. Technical report, Mozart Consortium, December 2001. Available at `http://www.mozart-oz.org/`.

[131] Peter Van Roy, Seif Haridi, Per Brand, and Gert Smolka. Three moves are not as bad as a fire. In *Workshop on Internet Programming Languages, International Conference on Computer Languages (ICCL 98)*, Chicago, IL, USA, May 1998.

[132] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.

[133] Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4):365–396, December 1986.

[134] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *16th Symposium on Operating System Principles*, October 1997.

[135] Joachim P. Walser. Feasible cellular frequency assignment using constraint programming abstractions. In the *1st Workshop on Constraint Programming Applications, CP 96*, August 1996.

[136] David H. D. Warren. *Applied Logic–Its Use and Implementation as a Programming Tool*. PhD thesis, University of Edinburgh, 1977. Reprinted as Technical Note 290, SRI International.

[137] D. Weyns, E. Truyen, and P. Verbaeten. Distributed threads in java, 2002.

[138] Claes Wikström. Distributed programming in Erlang. In the *1st International Symposium on Parallel Symbolic Computation* (*PASCO 94*), pages 412–421, Singapore, September 1994. World Scientific.

[139] Floyd Wilder. *A Guide to the TCP/IP Protocol Suite*. Artech House, Norwood, MA, 1998.

Swedish Institute of Computer Science
SICS Dissertation Series

01: Bogumil Hausman, *Pruning and Speculative Work in OR-Parallel PROLOG*, 1990.
02: Mats Carlsson, *Design and Implementation of an OR-Parallel Prolog Engine*, 1990.
03: Nabiel A. Elshiewy, *Robust Coordinated Reactive Computing in SANDRA*, 1990.
04: Dan Sahlin, *An Automatic Partial Evaluator for Full Prolog*, 1991.
05: Hans A. Hansson, *Time and Probability in Formal Design of Distributed Systems*, 1991.
06: Peter Sjödin, *From LOTOS Specifications to Distributed Implementations*, 1991.
07: Roland Karlsson, *A High Performance OR-parallel Prolog System*, 1992.
08: Erik Hagersten, *Toward Scalable Cache Only Memory Architectures*, 1992.
09: Lars-Henrik Eriksson, *Finitary Partial Inductive Definitions and General Logic*, 1993.
10: Mats Björkman, *Architectures for High Performance Communication*, 1993.
11: Stephen Pink, *Measurement, Implementation, and Optimization of Internet Protocols*, 1993.
12: Martin Aronsson, *GCLA. The Design, Use, and Implementation of a Program Development System*, 1993.
13: Christer Samuelsson, *Fast Natural-Language Parsing Using Explanation-Based Learning*, 1994.
14: Sverker Jansson, *AKL - - A Multiparadigm Programming Language*, 1994.
15: Fredrik Orava, *On the Formal Analysis of Telecommunication Protocols*, 1994.
16: Torbjörn Keisu, *Tree Constraints*, 1994.
17: Olof Hagsand, *Computer and Communication Support for Interactive Distributed Applications*, 1995.
18: Björn Carlsson, *Compiling and Executing Finite Domain Constraints*, 1995.
19: Per Kreuger, *Computational Issues in Calculi of Partial Inductive Definitions*, 1995.
20: Annika Waern, *Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction*, 1996.
21: Björn Gambck, *Processing Swedish Sentences: A Unification-Based Grammar and Some Applications*, June 1997.
22: Klas Orsvärn, *Knowledge Modelling with Libraries of Task Decomposition Methods*, 1996.
23: Kia Höök, *A Glass Box Approach to Adaptive Hypermedia*, 1996.
24: Bengt Ahlgren, *Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption*, 1997.
25: Johan Montelius, *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages, May*, 1997.
26: Jussi Karlgren, *Stylistic experiments in information retrieval*, 2000
27: Ashley Saulsbury, *Attacking Latency Bottlenecks in Distributed Shared Memory Systems*, 1999.

28: Kristian Simsarian, *Toward Human Robot Collaboration*, 2000.
29: Lars-Åke Fredlund, *A Framework for Reasoning about Erlang Code*, 2001.
30: Thiemo Voigt, *Architectures for Service Differentiation in Overloaded Internet Servers*, 2002.
31: Fredrik Espinoza, *Individual Service Provisioning*, 2003.
32: Lars Rasmusson, *Network capacity sharing with QoS as a financial derivative pricing problem: algorithms and network design*, 2002.
33: Martin Svensson, *Defining, Designing and Evaluating Social Navigation*, 2003.
34: Joe Armstrong, *Making reliable distributed systems in the presence of software errors*, 2003.
35: Emmanuel Frecon, *DIVE on the Internet*, 2004.
36: Rickard Cöster, *Algorithms and Representations for Personalised Information Access*, 2005
37: Per Brand, *The Design Philosophy of Distributed Programming Systems: the Mozart Experience*, 2005
38: Sameh El-Ansary, *Designs and Analyses in Structured Peer-to-Peer Systems*, 2005