The Limits of Network Transparency in a Distributed Programming Language

Raphaël Collet

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

December 2007

Faculté des Sciences Appliquées
Département d'Ingénierie Informatique
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis committee:

Yves Deville (chair)	UCL/INGI, Belgium
Marc Lobelle	UCL/INGI, Belgium
Per Brand	SICS, Sweden
Joe Armstrong	Ericsson, Sweden
Peter Van Roy (advisor)	UCL/INGI, Belgium

Abstract

This dissertation presents a study on the extent and limits of *network transparency* in distributed programming languages. This property states that the result of a distributed program is the same as if it were executed on a single computer, in the case when no failure occurs. The programming language may also be *network aware* if it allows the programmer to control how a program is distributed and how it behaves on the network. Both aim at simplifying distributed programming, by making non-functional aspects of a program more modular.

We show that network transparency is not only possible, but also *practical*: it can be efficient, and smoothly extended in the case of partial failure. We give a proof of concept with the programming language Oz and the system Mozart, of which we have reimplemented the distribution support on top of the Distribution Subsystem (DSS). We have extended the language to control which distribution algorithms are used in a program, and reflect partial failures in the language. Both extensions allow to handle non-functional aspects of a program without breaking the property of network transparency.

Acknowledgments

I want to thank all the people that have supported me during those nine years. This thesis took a long time to mature, and would not have been possible without the following people.

I am grateful to my advisor Peter Van Roy, who introduced me to Mozart and its distributed protocols, and trained me as a researcher. His constant enthusiasm has pushed me to complete this work.

I want to thank all my colleagues at the university. In particular, the people with whom I have explored the distribution of Oz: Boris Mejías, Yves Jaradin, Valentin Mesaros, Donatien Grolaux, Kevin Glynn, Iliès Alouini; and people who experienced constraint programming in Oz with me: Fred Spiessens, Luis Quesada, Stefano Gualandi, Renaud de Landtsheer, and Isabelle Dony. I also thank all the people who worked with me, and who I forgot to list here.

I am also grateful to the people of the Mozart consortium, in particular Per Brand, Erik Klintskog, Konstantin Popov, and Christian Schulte. They helped me when I needed some guidance to modify the virtual machine and the DSS. I also thank all the scientists I have met during these years, and with whom I learned so much.

This research has been supported by the projects PIRATES (Walloon Region), PEPITO (European IST FET Global Computing), EVERGROW (European 6th Framework Programme), SELFMAN (European 6th Framework Programme), and CoreGRID (European Network of Excellence on Grid and Peer-to-Peer technologies). I thank all the institutions that funded those projects.

Because my life has changed since I began playing music, I would like to thank all the musicians that made some great music with me. The following bands have been an immense opportunity to broaden my horizons: Glycerinn (with Jack, Sarah, and Felipe), the Confused Deputies (with Boris and Fred), and the Yellows (with Jean-François, Marc, Jérôme, Alexandre, and Christophe).

My last thanks go to my family, my parents, my brother, my sister with her husband and two little boys.

Contents

Abstract i

Acknowledgments iii

Contents v

1 Introduction 1
- 1.1 Distributed systems . . . 2
- 1.2 Models of distributed programming . . . 3
- 1.3 Network transparency . . . 5
- 1.4 Thesis and contributions . . . 7
- 1.5 Structure of the document . . . 9

2 An introduction to Oz 11
- 2.1 The kernel language approach . . . 11
- 2.2 Declarative kernel language . . . 12
  - 2.2.1 The store . . . 13
  - 2.2.2 Declarative statements . . . 13
  - 2.2.3 A few convenient operations . . . 16
- 2.3 Nondeclarative extensions . . . 16
  - 2.3.1 Exceptions . . . 16
  - 2.3.2 Read-only views . . . 17
  - 2.3.3 State . . . 18
- 2.4 Syntactic convenience . . . 18
  - 2.4.1 Declarative programming . . . 19
  - 2.4.2 Message passing . . . 21
  - 2.4.3 Stateful entities . . . 22
- 2.5 Distribution . . . 24
  - 2.5.1 Application deployment . . . 24

3 Application structure and distribution behavior 27
- 3.1 Layered structure . . . 27
  - 3.1.1 Using the declarative model . . . 28

3.1.2 Using message passing . . . 29
3.1.3 Using shared state concurrency . . . 31
3.2 Classification of language entities . . . 32
3.2.1 Mutable entities . . . 32
3.2.2 Monotonic entities . . . 33
3.2.3 Immutable entities . . . 33
3.3 Annotations . . . 34
3.3.1 Annotations and semantics . . . 35
3.3.2 Annotation system . . . 35
3.3.3 Partial and default annotations . . . 36
3.3.4 Access architecture . . . 36
3.3.5 State consistency protocols . . . 37
3.3.6 Reference consistency protocols . . . 38
3.4 Related work . . . 39
3.4.1 Erlang . . . 39
3.4.2 Java RMI . . . 40
3.4.3 E . . . 40

4 Asynchronous failure handling 43
4.1 Fault model . . . 44
4.1.1 Failures . . . 44
4.1.2 Failure detectors . . . 45
4.1.3 Entity fault states . . . 46
4.1.4 Concrete interpretation of fault states . . . 47
4.2 Failure handlers . . . 49
4.2.1 Definition . . . 49
4.2.2 No synchronous handlers for Oz . . . 49
4.2.3 Entity fault stream . . . 50
4.2.4 Discussion . . . 52
4.3 Making entities fail . . . 52
4.3.1 Global failure . . . 53
4.3.2 Local failure . . . 53
4.4 Failures and memory management . . . 54
4.4.1 Blocked threads and fault streams . . . 54
4.4.2 Entity resurrection . . . 55
4.5 Related work . . . 56
4.5.1 Java RMI . . . 56
4.5.2 Erlang . . . 56
4.5.3 The first fault model of Mozart . . . 56

5 Applications 59
5.1 Distributed lazy producer/consumer . . . 59
5.1.1 A bounded buffer . . . 60
5.1.2 A correct bounded buffer . . . 61
5.1.3 An adaptive bounded buffer . . . 62

5.1.4 A batch processing buffer . . . 63
5.2 Processes à la Erlang . . . 64
5.3 Failure by majority . . . 67
5.3.1 Algorithm . . . 67
5.3.2 Correctness . . . 68
5.3.3 The whole code of processes . . . 69
5.3.4 Variants . . . 69

6 Language semantics 75
6.1 Full language to kernel language . . . 75
6.2 Basics of the semantics . . . 79
6.2.1 The store . . . 79
6.2.2 Structural rules . . . 81
6.3 Declarative subset of the language . . . 82
6.3.1 Sequential and concurrent execution . . . 82
6.3.2 Variable introduction . . . 82
6.3.3 Unification . . . 83
6.3.4 Conditional statements . . . 85
6.3.5 Names and procedures . . . 86
6.3.6 By-need synchronization . . . 86
6.4 Nondeclarative extensions . . . 87
6.4.1 Nondeterministic wait . . . 87
6.4.2 Exception handling . . . 88
6.4.3 Read-only views . . . 89
6.4.4 State . . . 90

7 Distributed semantics 93
7.1 Reflecting network and site behavior . . . 94
7.1.1 Locality . . . 94
7.1.2 Network failures . . . 94
7.1.3 Site failures . . . 95
7.2 Reflecting entity behavior . . . 95
7.2.1 Entity failures . . . 96
7.2.2 Entity annotations . . . 96
7.3 Declarative kernel language . . . 98
7.3.1 Purely local reductions . . . 98
7.3.2 Variable introduction and binding . . . 98
7.3.3 Procedure creation and copying . . . 98
7.3.4 By-need synchronization . . . 99
7.4 Nondeclarative extensions . . . 99
7.4.1 Exception handling and read-only views . . . 100
7.4.2 State . . . 100
7.5 Failure handling . . . 102
7.5.1 Failure detectors . . . 102
7.5.2 Making entities fail . . . 104

7.6 Mapping distributed to centralized configurations 105
7.6.1 The mapping 105
7.6.2 Network transparency 105

8 Implementation **107**
8.1 Architecture of Mozart/DSS 107
8.2 The Distribution Subsystem 109
8.2.1 Protocols for mutables 110
8.2.2 Protocols for immutables 114
8.2.3 Protocols for transients 115
8.2.4 Handling failures 116
8.2.5 Distributed garbage collection 117
8.3 The language interface 118
8.3.1 Distributed operations in general 118
8.3.2 Distributed immutables 120
8.3.3 Remote invocations and thread migration 120
8.3.4 Unification and by-need synchronization 121
8.3.5 Fault stream and annotations 122
8.3.6 Garbage collection 123

9 Evaluation **129**
9.1 Ease of programming 129
9.2 Performance 129
9.2.1 Mozart/DSS vs. Mozart 130
9.2.2 Comparing protocols 131

10 Conclusion **135**
10.1 Achievements 135
10.2 Future directions 136

A Summary of the model **139**
A.1 Program structure 139
A.2 Failure handling 140

Bibliography **141**

1

Introduction

This dissertation presents a study on the extent and limits of network transparency in distributed programming languages. A programming language is said to be *network transparent* if a distributed program gives the same result as if it were executed on a single computer, provided network delays are ignored and no network failure occurs. The language is said to be *network aware* if the language definition allows to predict and control how the program is distributed, and its network behavior. The conjunction of both properties aims at simplifying distributed programming by separating a program's functionality, in which distribution can be ignored, from its distribution behavior, which includes network performance, partial failure (when part of the system fails), and security.

Earlier works have shown that network transparency is possible, like [Jul88]. We show that network transparency is also *practical*: it can be efficient, and smoothly extended in the case of partial failure. Efficiency is possible if the programming language supports programming with asynchronicity, which is reasonable in general, and fits well with distribution. Performance can also be tuned by the choice of distributed algorithms used by the underlying system, without affecting functionality in the case when there are no failures. Partial failure can be reflected in the language in a simple way, so that fault tolerance can be added in a modular fashion completely within the language. Security is beyond the scope of this thesis and is a subject of future work. We give a proof of concept with the programming language Oz and the system Mozart [Moz99]. We have extended the language to improve its network awareness, both for controlling distribution and handling partial failures.

This work is a continuation of earlier works on distributed programming languages, mainly done at the Swedish Institute of Computer Science (SICS). Among the results of those works are the system Mozart and the Distribution Subsystem (DSS), and two dissertations:

- Per Brand, in *The Design Philosophy of Distributed Programming Systems: the Mozart Experience* [Bra05], presents the first design, implementation, and evaluation of the distributed system Mozart.

- Erik Klintskog, in *Generic Distribution Support for Programming Systems* [Kli05], presents the design, implementation, and evaluation of the DSS, a middleware which provides efficient distribution support for programming languages.

Per Brand showed that asynchronous stream communication can be orders of magnitude faster than synchronous communication (such as Java RMI). He also showed that an Oz program was almost unchanged when going from centralized to distributed, and much simpler than a corresponding Java program.

This thesis both extends and simplifies the network-transparent distribution in Mozart. We have modified and extended the language Oz, in order to improve the network awareness in the language. We have reimplemented the distribution layer in the platform Mozart on top of the DSS middleware, and completed the latter to make it able to handle and reflect partial failures. We have also redesigned failure handling in Oz to make it completely asynchronous, and showed that it was the right default.

1.1 Distributed systems

Distributed systems are becoming ubiquitous; today all computers are connected to the internet, which provides many collaborative tools and programs. Moreover, many computers today contain multiple processors that run in parallel. This is a consequence of the current limits in increasing processors' speed, which makes manufacturers increase the number of processors instead. Computers with multiple processors can be considered as distributed systems on their own, with fast communication between processors.

Software development is progressively shifting towards concurrent and distributed programs that can take advantage of this available parallelism. Sequential programming is still acceptable for small programs, but not for large applications. By necessity, large programs will be distributed, and therefore concurrent. Alas, the introduction of concurrency into existing systems is rather poor, and inter-thread communication is often based on shared state. This model is difficult and bug-prone for programmers, which are discouraged to program in concurrent style. However, some systems propose different models for concurrent programming, like message passing in Erlang, and dataflow concurrency in Mozart.

Partial failures also make distributed programs more complex than centralized ones. Programs that run on a certain number of machines should be able to deal with faults in parts of the system. On one hand, writing distributed applications without taking partial failures into account was quickly seen as unrealistic. On the other hand, one needs abstractions to avoid the application

code to be cluttered with failure-handling code everywhere. Failure handling should be as modular as possible.

Programming languages and systems. We claim that the design of the programming language is essential when writing such applications. Extending a sequential language may lead to bad surprises, because the distributed program will *not* be sequential. So the language or its libraries should at least provide good abstractions to handle concurrency. Moreover, letting parts of a program share data introduces many subtle problems. For instance, remote references may be provided either via proxy entities, or transparently by the language itself. The programmer needs clear indication about what can be shared between sites[1]. Also, transferring data requires *serialization*.

There are basically two ways for a programming language to support the development of distributed applications. The first approach is to augment the language with libraries for distribution. Those libraries provide abstractions to make sites interact with each other. This typically involves communication channels, abstract representations of distributed entities, and so on. The programmer is responsible to integrate its application with the distribution library.

The second approach is to provide distribution as an inherent property of the programming language itself. In this case, we talk about a *distributed programming language*. A program is seen as a collection of threads and data that are spread over a set of sites. From a functional point of view, the interaction between the parts of the application is not different from the interaction between concurrent threads on a single host. However, the semantics of the language is extended to incorporate new aspects, like network latency and partial failures. This thesis explores some of the possibilities that are offered by these systems.

1.2 Models of distributed programming

The choices made to bring distribution in a programming language clearly determines the *model* that the programmer has to use. We define the programming model as a set of language constructs together with how they are executed [VH04]. It is sometimes called more informally *programming paradigm*. Examples are: declarative programming, object-oriented programming, processes with message passing. Here we are interested in the underlying models of the programming systems (the languages and their libraries) that are used to build distributed applications. We consider a few concepts that may or may not be part of a programming model.

[1] The *site* is the unit of localization. Sites execute code concurrently, and are independent of each other. A typical example is a system process.

Concurrency. By definition, a distributed program involves several activities that run more or less independently on different sites. This implies that the programming model is necessary concurrent and non-deterministic, because those properties are intrinsic to distributed systems. Therefore concurrent programming languages have an advantage over sequential languages when it comes to distribution. A concurrent language makes it possible to test a distributed program in a single site.

Moreover, good language support for controlling concurrency is also an advantage. For instance, in Oz one can synchronize two threads with a dataflow variable. This technique is simple, elegant, and is useful even if the threads are located on different sites.

Synchronous and asynchronous operations. Many programming languages only provide synchronous operations in their model. Synchronous operations are fast and natural in centralized applications, but they can be pretty slow in a distributed setting. Indeed, distributed synchronous operations often require several sites to exchange messages, and cannot proceed immediately. Asynchronous operations do not wait: the operation terminates immediately, and its effect will be performed eventually. This scheme fits well in a distributed environment: the operation may simply prepare a message and terminate, while the message delivery will perform the expected effect. The network latency is partly hidden to the user.

Most programming languages designed with distribution in mind provide asynchronous operations in their model [Van06]. In some of them, like Erlang, synchronous operations are not even part of the core of their model, but are simply defined as a derived concept [AWWV96].

Stateful and stateless data. Does the programming model makes a distinction between stateful and stateless data? This is more important than it seems at first sight. "One size fits all" does not hold for distributed data. On one hand, stateless data can be copied between sites, which provides minimum latency for operations. Once the data are copied, all operations are purely local.

On the other hand, stateful data need different protocols to handle their state and keep it consistent between sites. The state may be stationary, and behave like a server for remote operations, like distributed objects in Java Remote Method Invocation (RMI) [GJS96, Sun97]. But other protocols are useful as well. A migratable state may give better performance when a site has to perform many operations on it. Once the state has moved to the site, it can be considered as a cache, because several operations on that site may be performed in a batch. Replicating the state is yet another option, if read operations are more frequent than updates.

Multiplicity of paradigms. The ability to choose between several paradigms when writing a distributed application may lead to better programs. If a problem has a natural solution in a given programming model, its implementation will be simpler. The programmer may also choose the paradigm depending on the problem to solve *and* how various concepts of the language are distributed. The system does not force the programmer to emulate a distributed protocol on top of inappropriate concepts.

Distributed and local references. In programming systems that provide distribution as a separate library, distributed and local references often have different interfaces. An example is given by Java RMI, where distributed objects introduce exceptions where equivalent local objects would not. Distributed objects have a slightly different semantics, too. Reference integrity of remote objects is not guaranteed in general, for instance [Sun97]. Turning local objects into distributed ones may break an application.

Making no visible difference between local and distributed data allows the runtime system to choose the right representation for a given datum. A local reference that is sent to a remote site automatically switches to a distributed representation for the corresponding data. The conversion may be reverted once the distributed reference is used by one site only. This implies less effort from the programmer.

Partial failures. They are inherent to distributed systems, so reflecting failures in the programming model is very important. It basically provides the programmer with a semantic representation of the failure. This semantic support allows the programmer to *reason* about failures, and handle them properly. Besides a semantic representation, the programming model should also provide a way to detect failures. Failure detectors are the basic ingredient of failure handlers.

We believe that causing a partial failure by program can be useful: it may simplify a failure handler. Sometimes a component cannot be fixed easily because it strongly depends on another component, which has failed. Making the former one fail may accelerate the failure recovery, which can be handled at a higher level in the application.

1.3 Network transparency

As we said, network transparency states that the result of a distributed program is the same as if it were executed on a single computer, in the case when no failure occurs. The meaning can be more precise if we consider network transparency at the level of the programming language. A given entity or piece of code is network transparent if its semantics does not depend on whether it is run in a distributed environment, provided no failure occurs.

Several forms of transparency have been proposed in the literature. The following ones are taken from [ISO98]. They use the term *resource* in a very general sense. When applied to a programming language, a resource typically corresponds to an object or an agent.

- *Access transparency* masks differences in data representation and invocation mechanisms, to provide a single and uniform access to resources.
- *Location transparency* states that the user of a resource should not be aware of where the resource is physically located. *Migration transparency* states that the user should not be aware of whether a resource or computation has the ability to move to a different location, while *relocation transparency* guarantees that its migration should not be noticeable to the user.
- *Replication transparency* makes a resource appear as unique even if it is replicated among several locations. *Persistence transparency* makes no difference between resources located in volatile and permanent memory.
- *Failure transparency* masks the failure and possible recovery of resources or computations.
- *Transaction transparency* masks coordination of activities amongst a configuration of entities to achieve consistency.

Our definition of network transparency covers the above notions of access, location, migration, relocation, and replication transparency. We also cover transaction transparency in the sense that primitive operations on distributed entity should be atomic, just like in the centralized case. Persistence transparency is rarely present in programming languages, where the program's memory is often considered as volatile. Our definition does not cover failure transparency, and our proposal for a distribution model in Chapter 4 will even make failures explicit in the language.

In practice. Some researchers have maintained that network transparency cannot be made practical, see, e.g., Waldo *et al.* [WWWK94]. They cite four reasons: pointer arithmetic, partial failure, latency, and concurrency. The first reason (pointer arithmetic) disappears if the language has an abstract store. The second reason (partial failure) requires a reflective fault model, which we designed for the Distributed Oz language. The authors of the paper above expected that failures could be always hidden behind abstractions. They were wrong: sometimes failures cannot be resolved locally, and requires some global action.

The final two reasons (latency and concurrency) lead to a layered language design. Latency is a problem if the language relies primarily on synchronized operations, like procedure calls. The authors of the paper explicitly mention the disappointing experience of remote procedure calls, that they see as the

only way to make distributed objects. In the terminology of Cardelli, latency is a network awareness issue [Car95]. The solution is that the language must make asynchronous programming both simple and efficient.

Concurrency is also seen as an obstacle. A closer look at the paper reveals that the authors actually talk about *shared-state concurrency*. Indeed, most people consider that programming languages are always stateful. The problem with concurrency in those languages is how to control concurrent accesses to state, in order to avoid invariant violations and glitches. A solution is to support a form of *stateless concurrency*, known as *dataflow concurrency*. Concurrent threads interact by sharing values, and automatically synchronize on the availability of data. Threads can be programmed as if they were never waiting for data. An example of this kind of communication is pipelining in Unix-like systems. Moreover, values can be copied between memory stores, which substantially reduce their latency. Using dataflow concurrency can reduce the need for shared state to a minimum. We conclude that language design is an important issue for network transparency.

1.4 Thesis and contributions

This thesis proves that network transparency is practical in a distributed programming language. It gives concrete proposals of language extensions that deal with performance and failure handling, and demonstrates their usage with practical examples. It also describes the implementation of the platform Mozart/DSS, with insights on various implementation issues.

Contributions. This thesis extends, simplifies, and completes the past work on network-transparent distribution in Mozart. The initial distribution model and the initial failure detection model [HVS97, VHB99] formed the core of the first distributed release of Mozart in 1999. Erik Klintskog made the first design of a distribution subsystem (DSS) in which the distribution support is completely factored out of the run-time emulator [Kli05]. The work on the DSS was incomplete, however. The present thesis brings the following scientific and technical contributions.

- We extend the distribution model of Oz to make it customizable. We introduce entity annotations, so that the programmer has the ability to choose between several protocols for each entity, including its distributed memory management.
- We design a failure handling model for Oz that is simpler and more expressive than the initial one. Each language entity produces a stream of fault states that is extended asynchronously, whenever the entity's fault state changes.
- We design an effective post-mortem finalization mechanism based on the fault stream. This mechanism did not exist in the language.

- We give distributed programming patterns that show how the system simplifies programming robust distributed systems.

- We complete Erik Klintskog's work by presenting more precise definitions of the distribution protocols that include failure handling, in particular the mobile state protocol.

- We have rebuilt the distribution support of the platform Mozart on top of the DSS library, and implemented the new distributed programming model of the language. The reflection of failures in the language, and the implementation of the new language features (annotations, fault stream, making entities fail) are entirely our work.

- We have also completed the implementation of the DSS. In particular we have rewritten all entity protocols such that they can handle partial failures. We have also extended the DSS interface to handle and reflect entity failures.

- We evaluate the new implementation in a realistic setting.

Publications. The following publications contain substantial contributions by the author on the topics of this thesis. The first two papers focus on the extension of a mobile state protocol to make it handle failures. That protocol is part of the platform Mozart. The implementation of that protocol is now part of the DSS. Its semantics as a migratory protocol are given in Chapter 7, and its implementation is described in Chapter 8.

- Peter Van Roy, Per Brand, Seif Haridi, and Raphaël Collet. A lightweight reliable object migration protocol. *Lecture Notes in Computer Science*, 1686:32–46, 1999 [VBHC99].

- Per Brand, Peter Van Roy, Raphaël Collet, and Erik Klintskog. Path redundancy in a mobile-state protocol as a primitive for language-based fault tolerance. Research Report RR2000-01, Université catholique de Louvain, Département INGI, 2000 [BVCK00].

The next two papers propose a formal definition of lazy computations in terms of concurrent constraints. That definition led to an efficient distributed implementation of that concept. Laziness is mentioned in Chapter 3, and a concrete example of its usage is shown in Chapter 5. Its semantics are defined in Chapters 6 and 7, and its implementation is described in Chapter 8.

- Alfred Spiessens, Raphaël Collet, and Peter Van Roy. Declarative laziness in a concurrent constraint language. 2nd International Workshop on Multiparadigm Constraint Programming Languages MultiCPL'03, 2003 [SCV03].

- Raphaël Collet. Laziness and declarative concurrency. 2nd Workshop on Object-Oriented Language Engineering for the Post-Java Era: Back to Dynamicity PostJava'04, 2004 [Col04].

The fifth paper shows how to design a transactional system for a distributed store of objects, on top of an overlay network. That work put some emphasis on what kind of primitives were desired in the language to handle failures.

- Valentin Mesaros, Raphaël Collet, Kevin Glynn, and Peter Van Roy. A transactional system for structured overlay networks. Research Report RR2005-01, Université catholique de Louvain, Département INGI, 2005 [MCGV05].

The latter paper is the author's proposal to favor asynchronous failure handling in a distributed programming language. The paper contains the essential contributions of Chapter 4.

- Raphaël Collet and Peter Van Roy. Failure handling in a network-transparent distributed programming language. In C. Dony et al., editor, *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 121–140. Springer-Verlag, 2006 [CV06].

1.5 Structure of the document

Chapter 2 gives an introduction to the programming language Oz. That language is the vehicle we have chosen to explain all our proposals. This chapter may safely be skipped if the reader already knows the language. Note that most code snippets in Oz appear in special figures called "snippets".

Chapters 3 and 4 detail our proposals for dealing with an application's distributed behavior, and failure handling, respectively. The programming model is exposed and explained, together with some practical intuition for all concepts. Concrete examples using those language features are given in Chapter 5.

Chapters 6 and 7 propose a formal definition of the language, in centralized and distributed settings, respectively. Operational semantics are given for the core of the language, and the centralized semantics are refined into distributed semantics that reflect the aspects related to distribution.

Chapter 8 describes the implementation of Mozart/DSS, the reimplementation of Mozart on top of the DSS library. It gives a definition of the protocols that are used to implement basic operations on distributed entities, sketches the DSS application programming interface, and explains how the distribution support is implemented on top of it.

Chapter 9 evaluates the work done so far. Comparisons with other systems are made. Chapter 10 concludes the work. Scientific results are summarized,

and future directions are given. Appendix A gives a summary of the model, and the language extensions proposed in this work.

2

An introduction to Oz

In this text, we use the programming language Oz as a vehicle to express a certain number of concepts related to distributed programming. The concepts themselves are language independent, but few programming languages are able to express them in a natural way. Oz makes it possible, thanks to its support for multiple programming paradigms, among which we find declarative programming, dataflow concurrency, and object-oriented programming [Smo95, VH04]. This chapter gives a quick introduction to the language, and the basic model of its distribution. A formal definition of the language is given in Chapter 6.

2.1 The kernel language approach

The language Oz is based on a small set of concepts, that form a *kernel language*, called Kernel Oz. In the kernel language, all concepts are primitive: they cannot be defined in terms of each other. The full language is defined as the kernel language extended with *language abstractions*, i.e., programming abstractions with syntactic support. All those abstractions are defined in terms of Kernel Oz, so that every program can be reduced to an equivalent program in the kernel language.

The advantage of the kernel language approach is that the language is defined by layers. The bottom layer is the kernel, with all primitive concepts. The upper layers then define abstractions built on concepts defined in layers below. In practice, two layers are enough to define a very expressive language. Figure 2.1 on the following page shows a certain number of concepts that are present in the language. All concepts in bold font are part of the kernel language, while the others are derived concepts. The arrows indicate from what a given concept derives.

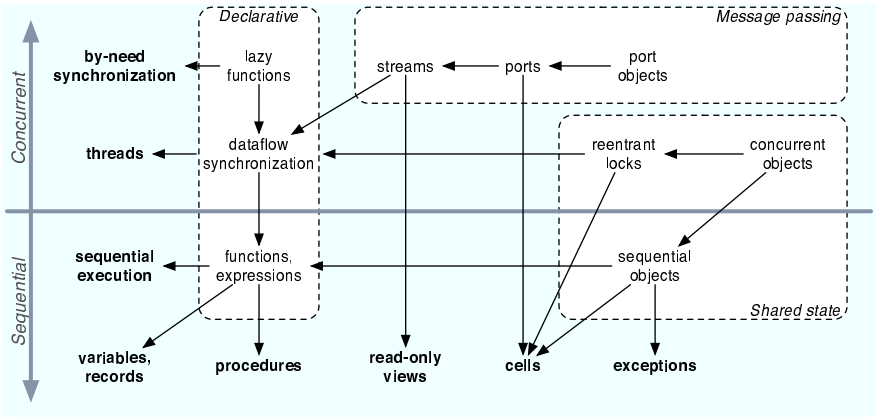


Figure 2.1: Primitive and derived concepts in the language Oz

The paradigm space. When writing a program or module, the programmer will often use only a subset of the concepts provided by the programming language. Every subset of concepts defines a *programming model*, or *paradigm*. Each paradigm comes with its own set of techniques and design rules. As you can see, the derived concepts shown in figure 2.1 are not placed randomly. There are two major axes in the diagram.

The base of all derived concepts is a sequential declarative language, which can be made a functional language with the appropriate language support. It contains no state and no concurrency. The language is mainly enriched by adding state (right direction in the diagram) and concurrency (upper direction). Moreover, the derived concepts are grouped together to form three major paradigms, namely declarative, message passing, and shared state programming.

Section 2.2 introduces the kernel concepts of the core language, together with concurrency. This part of Oz is completely declarative. The non-declarative kernel concepts are introduced in section 2.3. In section 2.4 we present syntactic extensions, and language abstractions used in the text. There we show two important techniques for controlling concurrency in the presence of state: message passing with *ports*, and *locks*. Note that we have chosen to introduce concurrency before state in the presentation. This is because it is possible to make distributed programs that are declarative; all they need is concurrency. The base distribution model is given in Section 2.5.

2.2 Declarative kernel language

An Oz program consists in a set of *threads* computing over a shared *store*. Threads execute statements in a sequential way. The program will be sequential if it contains a single thread. The store is the memory of the program; it

contains logic variables and data structures used by the threads. Threads are connected to the store by variables.

2.2.1 The store

The program store has two parts: the so-called *constraint store* and the *procedure store*. The constraint store contains logic statements about the program's variables. For now consider that those logic statements have the form $x{=}t$, where x is a variable, and t is either a variable or a value. The procedure store contains the procedures created by the program.

Values. The different kind of values are *integers*, *atoms*, *names*, and *records*. Names are primitive entities that have no structure; they are used to give an identity to other data structures, like procedures. The boolean values **`true`** and **`false`** are defined as names.

Atoms are literal constants that are defined by a sequence of characters. Syntactically they are words starting with lowercase letter, like `atom` or `nil`, or they are surrounded by quotes, like `´Hello world´` or `´|´`. A *literal* is either a name or an atom, and a *simple value* is either a literal or an integer.

A *record* is a compound value $l(k_1{:}v_1 \ldots k_n{:}v_n)$ formed by a *label* l (a literal) and *fields* $v_1, \ldots, v_n$. Each field v_i is associated to a key k_i, which is a simple value. An example is `person(name:raph age:32)`. A *tuple* is a record whose keys are the integers 1 to n. Syntactically the keys can be omitted, as in `person(raph 32)`, which is equivalent to `person(1:raph 2:32)`. Lists are defined in terms of tuples and atoms as follows. A list is either the empty list (the atom `nil`), or a head-tail pair `´|´`(xy). The latter can be written infix as x`|`y.

Variables. The store contains logic variables that can be bound to values and other variables. Upon creation, a variable x is always unbound. It is bound to a value v if the store contains the statement $x{=}v$. It can be bound to at most one value, and the binding cannot change over time. A variable bound to a value is said to be *determined*.

Variables x and y can also be bound together, if the store contains $x{=}y$. The binding relation is transitive: if x is bound to a value v, then y is also bound to the same value v. Note that undetermined variables can be bound together.

2.2.2 Declarative statements

The declarative kernel statements are given in Figure 2.2 on the following page.

S	::=	**skip**	*empty statement*
	|	$S_1 S_2$	*sequential composition*
	|	**local** X **in** S **end**	*declaration*
	|	$X{=}Y$ | $X{=}v$	*unification*
	|	**if** X **then** S_1 **else** S_2 **end**	*conditional statement*
	|	**proc** {P $X_1 \ldots X_n$} S **end**	*procedure creation*
	|	{P $X_1 \ldots X_n$}	*procedure application*
	|	**thread** S **end**	*thread creation*
	|	{`WaitNeeded` X}	*by-need synchronization*
v	::=	s	*simple value*
	|	$l(k_1{:}X_1 \ldots k_n{:}X_n)$	*record*
s	::=	l | i	*literal or integer*
l	::=	*atom* | **true** | **false**	*atom or name*
P, X, Y	::=	*identifier*	*variable identifier*

Figure 2.2: Declarative kernel concepts of Oz

Empty statement and sequential composition. The execution of the statement **skip** simply has no effect. The sequence of statements $S_1 S_2$ first executes S_1, then executes S_2.

Declaration. The statement **local** X **in** S **end** creates a fresh variable x in the store, and reduces to the statement S, where X is associated to x. It defines a *lexical scope* between the keywords **in** and **end** for the identifier X.

In order to be executable, a statement must have all its free identifiers correspond to store variables. For the sake of simplicity, in the rest of the text, we will refer to the variable corresponding to identifier X as "the variable X."

Unification. Variables are bound in the store by unification. The statements $X{=}Y$ and $X{=}v$ add the necessary variable bindings to make their arguments equivalent. For instance, the following statements binds `R` to the record `foo(a:X b:Y)`, then by unification of records makes both `X` and `Y` equal to `Z`, and finally binds them all to 42.

```
R=foo(a:X b:Y) R=foo(a:Z b:Z) Z=42
```

A unification triggers an error if its arguments cannot be made equal. The error shows up as an exception (see below).

Conditional statement. The statement **if** X **then** S_1 **else** S_2 **end** blocks until the variable X is determined. It reduces to S_1 if X equals **true**, and S_2

if X equals **`false`**. Other values trigger an error (in the form of an exception, see below).

Procedure creation. The statement **`proc`** `{`$P\ X_1 \dots X_n$`}` S **`end`** creates a fresh name ξ, adds the procedure $\lambda X_1 \dots X_n.S$ under the name ξ in the procedure store, and reduces to the statement $P{=}\xi$. Since the name ξ is fresh, the procedure store defines a mapping of names to procedures. Note also that the name makes the procedure a first-class entity: it can be passed as arguments and stored in data structures.

The procedure may refer to anything in the lexical scope of its definition. Those external references are determined by the variables corresponding to the free identifiers in S that do not occur in the procedure's parameters $X_1, \dots, X_n$. The **`proc`** statement defines a lexical scope for the identifiers $X_1, \dots, X_n$. Note that the declaration of the identifier `P` is done outside the procedure definition. This implies that the order of creation of procedures that use each other is irrelevant, provided they are all created before their use.

Procedure application. The statement `{`$P\ Y_1 \dots Y_n$`}` blocks until the variable P is determined. If P is the name of an n-ary procedure $\lambda X_1 \dots X_n.S$ in the procedure store, it reduces to the statement S where each X_i corresponds to the variable Y_i. If P is not a procedure, or a procedure with a different arity, the statement triggers an error.

Thread creation. The statement **`thread`** S **`end`** creates a new thread that consists of the statement S. The new thread is independent from the current thread. The threads in a program execute concurrently. The following primitive store operations are guaranteed to be *atomic*: creating a fresh name or variable, binding a variable, and storing a procedure. This makes the concurrency in Oz fine-grained.

A thread is *runnable* if its first statement is executable. When the first statement of the thread blocks on a variable, we say that the thread is *blocked* or *suspended*. It remains blocked until the variable is determined by the store. It then becomes runnable again.

By-need synchronization. We say that a variable X is *needed* when either it is determined, or a thread waits for its determination. This property is *monotonic*: when a variable is needed, it remains needed. The statement `{WaitNeeded` X`}` blocks until the variable X becomes needed. This primitive is used to attach a *lazy* computation to the variable X: a thread blocks until X becomes needed, then computes the value of X.

```
thread
   {WaitNeeded X}     % block until X is needed
   ...                % compute a value and assign it to X
end
```

2.2.3 A few convenient operations

Here are a few extra statements, that complete the declarative kernel language with common primitive operations.

Tests. The statement X=(Y`==`Z) blocks until there is enough information in the constraint store to logically entail Y=Z or $Y \neq Z$. The statement then reduces to X=**`true`** or X=**`false`**, respectively. The operator `\=` is similar, but returns the opposite results. The statement

$$X{=}Y \ op \ Z \qquad \text{with } op \in \{\texttt{<}, \texttt{=<}, \texttt{>}, \texttt{>=}\}$$

blocks until both Y and Z are determined to atoms or integers. It then reduces to X=**`true`** or X=**`false`**, depending on the result of the comparison. Atoms are ordered lexically, and atoms and integers are not comparable.

Arithmetic operations. The statements

$$X{=}\tilde{}Y \qquad X{=}Y \ op \ Z \qquad \text{with } op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \textbf{\texttt{div}}, \textbf{\texttt{mod}}\}$$

block until their arguments (Y and Z) are determined to integers, and reduce to X=i, where i is the result of the corresponding operation.

Record operations. The statements

`{Label` X Y`}` `{Width` X Y`}` `{Arity` X Y`}`

block until X is determined to be a record $l(k_1{:}v_1 \ldots k_n{:}v_n)$. They then reduce to Y=l, Y=n, and Y=k_1`|` $\ldots$ `|`k_n`|nil`, respectively.

The statement X=Y`.`Z blocks until Z is determined to a simple value and Y to a record $l(k_1{:}v_1 \ldots k_n{:}v_n)$. If Z is equal to k_i for a certain i, the statement reduces to X=v_i. Otherwise an exception is raised.

2.3 Nondeclarative extensions

Figure 2.3 on the next page shows nondeclarative extensions to the language. They include exceptions, state, and read-only views.

2.3.1 Exceptions

Try statement. The statement **`try`** S_1 **`catch`** X **`then`** S_2 **`end`** starts by executing S_1. If S_1 terminates normally, the statement reduces to **`skip`**. If an exception x "escapes" from S_1, i.e., x is raised inside S_1 but not caught, the **`try`** statement reduces to S_2 (the *exception handler*), with X corresponding to variable x. Note that the **`try`** statement can only catch exceptions raised in its own thread.

S	::=	**try** S_1 **catch** X **then** S_2 **end**	*try statement*
	|	**raise** X **end**	*raise statement*
	|	{FailedValue X Y}	*failed value creation*
	|	X=!!Y	*read-only view creation*
	|	{NewCell X Y}	*cell creation*
	|	X=Y:=Z	*cell exchange*

Figure 2.3: Nondeclarative kernel concepts of Oz

Raise statement. The statement **raise** X **end** throws an exception with the variable X. The exception will be caught by closest exception handler in the thread, if it exists.

Failed values. A *failed value* is a special value that encapsulates an exception. The statement {FailedValue X Y} creates a failed value v encapsulating the variable X, and reduces to Y=v. Any statement that attempts to use the value v automatically raises an exception with X.

Failed values are used to pass exceptions between threads. This is particularly useful when an exception occurs in a thread that is responsible to compute the value of a variable Z. If the thread binds Z to a failed value encapsulating the exception, all the other threads that use Z will know why the value of Z could not be determined.

```
thread
   try
      local V in
         ...                  % compute a value V
         Z=V                  % return the value in Z
      end
   catch E then
      {FailedValue E Z}       % return a failed value in Z
   end
end
```

2.3.2 Read-only views

"Bang bang" operator. The statement X=!!Y creates a *read-only view* u of variable Y, and reduces to X=u. A read-only view of a variable is logically equal to that variable, but it cannot be bound by the program; every statement that attempts to bind it blocks. When the variable is determined to a value, the read-only view is also bound to that value.

Read-only views are used to protect abstractions from accidental bindings that may break them. They are often used to build robust *streams*. A stream is a list that is built incrementally. During its construction, a read-only view

of its tail variable is put in the list tuple. This prevents the consumer of the stream to bind the tail, and provoke a failure in the producer of the stream.

2.3.3 State

The primitive concept of state in Oz is the *cell*. A cell is a mutable container for a variable. Cells are contained in a new part of the store called the *cell store*. A cell is a first-class value, identified by a name, and its contents is given by a (possibly determined) variable. Similarly to the procedure store, the cell store defines a mapping of names to variables.

A cell can be used by a single thread or shared among several threads. Concurrent accesses to a cell are relatively easy to synchronize, because read and write operations are combined in a single, atomic *exchange* operation. Concurrency control abstractions can be built with cells, like ports and locks (see Section 2.4).

Cell creation. The statement `{NewCell` X Y`}` creates a fresh name ξ, adds the pair ξ:X in the cell store, and reduces to the statement Y=ξ.

State exchange. The statement X`=`Y`:=`Z blocks until Y is determined. If Y is a cell ξ:w, the cell store is updated with ξ:Z, and the statement reduces to X=w.

For instance, if `C` is a cell that contains an integer, the following procedure adds `N` to the contents of the cell. Assume two threads update concurrently `C` with `AddCounter`. If the initial state is x, the first thread takes x and put a variable y in the cell, then the second thread takes y and put a variable z. The second thread computes the new state z with y, and automatically waits for the first thread to determine y. No race condition occurs if all state updates are done this way.

```
proc {AddCounter N}
   local X Y in
      X=C:=Y       % get contents X, and put new contents Y
      Y=X+N        % determine the value of Y
   end
end
```

2.4 Syntactic convenience

This section introduces syntactical convenience that corresponds to the full language, at least the part of the language that is relevant for this text. All the rules in this section suggest how to rewrite statements in the full language to statements in the kernel language.

Comments. All the characters that follow the percent sign (`%`) until the end of the line are comments.

2.4.1 Declarative programming

Declarations. Multiple variables can be declared simultaneously. For instance,

```
local X Y in S end     ⇒  local X in local Y in S end end
```

If a declaration statement comprises the body of a procedure definition or the branch of a conditional, **local** and **end** can be omitted. For example:

```
proc {P} Y in S end    ⇒  proc {P} local Y in S end end
```

Declaration can be combined with initialization through unification:

```
local X=5 in S end     ⇒  local X in X=5 S end
```

Expressions. We first define the statement Z=`{`$P\ X_1 \ldots X_n$`}` as shorthand for `{`$P\ X_1 \ldots X_n\ Z$`}`. Similarly, nesting of record construction and procedure application avoids declaration of auxiliary variables. For example:

```
X=b({F N+1})           ⇒  local Y Z in
                             Y=N+1 X=b(Z) {F Y Z}
                          end
```

Record construction is given precedence over procedure application to allow more procedure definitions to be tail recursive. The construction is extended analogously to other statements, allowing statements as expressions. For example:

```
X=local Y=2 in         ⇒  local Y=2 in X={P Y} end
     {P Y}
  end
```

Procedure definitions as expressions are tagged with a dollar sign (`$`) to distinguish them from definitions in statement position:

```
X=proc {$ Y} Y=1 end   ⇒  proc {X Y} Y=1 end
```

Another common expression is the *anonymous* variable:

```
_                      ⇒  local X in X end
```

Functions. Motivated by the functional notation of procedure calls, we define a function of n arguments as being equivalent to a procedure of n+1 arguments, the extra argument being bound to the result of the function body, which is an expression:

```
fun {Inc X} X+1 end    ⇒  proc {Inc X Y} Y=X+1 end
```

Lazy functions. Function definitions with the decoration `lazy` create a thread that uses `WaitNeeded` to wait for the result to be needed. Chapter 6 proposes a translation that avoids creating threads in the presence of tail recursive calls.

```
fun lazy {Inc X}          ⇒  proc {Inc X Y}
   X+1                          thread
end                                {WaitNeeded Y} Y=X+1
                                end
                             end
```

Lists. Complete lists can be written by enclosing the elements in square brackets. For example, `[1 2]` abbreviates `1|2|nil`, which abbreviates `´|´(1 ´|´(2 nil))`.

Infix tuples. The label `´#´` for tuples `´#´(X Y Z)` can be written infix: `X#Y#Z`.

Pattern matching. Programming with records and lists is greatly simplified by pattern matching. For instance, a pattern matching conditional

```
case X of person(name:N age:A) then S1 else S2 end
```

is an abbreviation for

```
if {Label X}#{Arity X} == person#[age name] then
   N A in X=person(name:N age:A) S1
else S2 end
```

The **else** part is optional and defaults to **else skip**. Multiple clauses are handled sequentially, for example:

```
case X                    ⇒  case X of f(Y) then S1 else
of f(Y) then S1                 case X of g(Z) then S2 end
[] g(Z) then S2              end
end
```

The **try** statements are also subject to pattern matching. For example:

```
try S1                    ⇒  try S1 catch Y then
catch f(X) then S2              case Y of f(X) then S2
end                             else raise Y end
                                end
                             end
```

Loops. Recursive functions are expressive enough to implement all kinds of loops in the language. Oz supports a simple yet powerful **for** loop on lists:

```
for X in L do S end    ⇒  {ForAll L proc {$ X} S end}
```

where `ForAll` is defined as

```
proc {ForAll L P}
   case L of X|T then {P X} {ForAll T P} else skip end
end
```

Waiting for determinacy. A common abstraction is the procedure `Wait`, that blocks until its argument is determined. It is often used to explicitly synchronize threads. It can be defined as follows, using the blocking behavior of `==`.

```
proc {Wait X} _=(X==1) end
```

2.4.2 Message passing

Ports. A port is associated to a stream (defined in Section 2.3.2 above), which lists all the messages sent on the port. Ports are defined by two operations: `NewPort`, which creates a port and its stream, and `Send`, which sends a message on a given port. They can be defined in terms of cells as

```
fun {NewPort S}
   T in S=!!T {NewCell T}
end
proc {Send P X}
   T in X|!!T=P:=T
end
```

Note, however, that ports are not truly defined like that. When it comes to distribution, they do not behave like cells, but have a behavior of their own. This is because ports are intrinsically asynchronous, which cells are synchronous.

Ports are very convenient to handle nondeterminism, since they are asynchronous (they never block), and all the messages sent to a port are serialized into a list (the stream).

Port objects. A *port object* consists in a port and a thread that reads sequentially its message stream. Because the message processor is sequentially reading a list, it can be written as a simple recursive function. The latter can use an accumulator to maintain a state.

Snippet 2.1 on the following page shows a simple abstraction that builds port objects. The argument function `Func` takes the object's current state and a message, and returns the new state of the object. The function `FoldL` is used to apply `Func` on every message. An example is shown below, with a object `Counter`. This object recognizes three kind of messages: `inc`, `inc(N)`, and `get(N)`. See how the latter binds `N` to the current value of the counter.

```
fun {NewPortObject Init Func}
   S in
   thread {FoldL S Func Init} end
   {NewPort S}
end
fun {FoldL L F I}
   case L of X|T then {FoldL T F {F I X}} else nil end
end

local
   fun {F Count Msg}
      case Msg
      of inc    then Count+1
      [] inc(N) then Count+N
      [] get(N) then N=Count Count
      end
   end
in
   Counter={NewPortObject 0 F}
end
{Send Counter inc(3)}     % increment counter by 3
```

Snippet 2.1: An abstraction to create port objects, and an example of a counter object

Active objects. Active objects are similar to port objects, except that they use a stateful object instead of a function to process the messages. This technique is pretty easy to work with, because the underlying object is used sequentially.

2.4.3 Stateful entities

State operations. The full language supports two simplified versions of the state exchange operation, to simply read and write the state.

```
X=@C          ⇒  X=C:=X
C:=X          ⇒  _=C:=X
```

Objects and classes. In Oz an *object* is defined as a unary procedure, that takes a *method* as its argument. The method is represented as a record, and is therefore first-class. An object usually maintains a proper state. A *class* is a value that creates objects. A precise kernel equivalent of classes, and their inheritance mechanism, is given in [VH04].

Snippet 2.2 on the next page illustrates the class syntax with an example with two classes and one object. A base class `Stack` defines an attribute `elements`, which is identified by an atom. It also defines four methods: `init`, `isEmpty`, `push`, and `pop`. The state operators are extended to attributes. The

```
class Stack
   attr elements       % list of elements, from top to bottom
   meth init
      elements:=nil            % initializer
   end
   meth isEmpty(B)
      B=(@elements==nil)
   end
   meth push(X)
      T in T=elements:=X|T     % put X in front of elements
   end
   meth pop(X)
      T in X|T=elements:=T     % extract front element
   end
end

class Stack2 from Stack        % Stack2 extends Stack
   meth top(X)
      {self pop(X)} {self push(X)}
   end
end

S={New Stack2 init}     % create an object of class Stack2
{S push(42)}            % call method push(42) of Obj
```

Snippet 2.2: Two classes and an object

class `Stack2` extends the class `Stack`. It defines a method `top`, which is implemented with the methods `push` and `pop` of the object, which is accessible by the keyword **`self`**. Then the function `New` is used to instanciate the class, and initialize the object.

There is no concurrency control by default in objects. Concurrent method invocations will be executed concurrently, and state updates are subject to the same kind of atomicity as cells. Objects often use *locks* to create critical sections inside methods.

Locks. A lock is a binary semaphore, which controls the access to the lock itself. At most one thread can be in a given lock. The only operation takes the lock, executes a statement, and releases the lock. If another thread already owns the lock, the operation blocks until the lock is available. The **`lock`** statement is translated as follows.

`lock` `L` **`then`** S **`end`** $\Rightarrow$ `{L proc {$}` S **`end`**`}`

The lock `L` is created by the following function, which implements a basic lock with a cell and a procedure.

```
fun {NewLock}
   C={NewCell unit}
in
   proc {$ P}
      X Y in X=C:=Y {Wait X} {P} Y=X
   end
end
```

When the lock is applied, it places a new variable into its cell, and waits until the former variable in the cell is determined. Once it is determined, the lock is available. The lock then executes the statement, which is abstracted by a nullary procedure. It then release the lock by binding the new variable to the value. Several threads applying the same lock will form a chain, and pass the value **unit** between each other via a shared variable. The cell's function is to connect a thread to its successor in the waiting queue.

The language Oz actually provides *reentrant locks*. Those locks permit a thread to take the same lock several times. This is useful when two procedures or methods call each other, and protect a shared state with the same lock. For a definition of reentrant locks in Kernel Oz, see [VH04]. Note that distinct locks are not connected to each other in any way; deadlocks are possible, and the language provides no deadlock detection mechanism.

2.5 Distribution

In Oz, a distributed program is usually defined as a centralized program where entities and threads would be partitioned among *sites*. One can also define a distributed program as a set of centralized programs running on their own sites, and sharing language entities. Both definitions are in fact valid and equivalent, because the language is *network-transparent*.

Most of the distribution is hidden to the programmer, as shared entities keep their semantics almost intact. What happens is that the programming system uses dedicated protocols to implement the entities' semantics. In Mozart the distribution of entities is designed to give the programmer full control over network communication patterns that occur because of language operations. Not all entities use the same protocol, every entity use a protocol that is adapted to its nature: mutable, immutable, or transient. This subject is discussed in detail in Chapter 3.

2.5.1 Application deployment

The deployment of an application covers two situations that are handled differently. The first one is how the distribution between sites that already share entities evolve. The second one is how to create new sites, or connect independent sites.

Sites that know each other. Sites that already share entities evolve by following which entities they share. They can share new entities by *transitivity*: a site obtains a reference to an entity via another entity that it already refer. For instance, if site a sends a value x on a port, and site b reads the message stream of that port, site b automatically has access to x.

Note that sites also connect to each other by transitivity: if sites a and b are connected together, and b and c as well, then a and c will automatically connect to each other, if the entities they share requires so. This depends on which protocols are used by the shared entities.

Sites can also reduce their set of shared references. This is handled by *distributed memory managment*. The system detects when a site no longer refers to a given entity, and can globally remove an entity from the distributed program. This always works, except for distributed reference cycles, i.e., reference cycles that involve several sites.

Connecting sites. The definitions we just gave of a distributed program suggests two ways of deploying an application over new sites: either by splitting a site into several sites, or by connecting distinct sites. Mozart uses the second approach, because it is easier to implement and to control in the program. This is provided by the module `Connection`.

The function call `{Connection.offer X}` returns a *ticket*, i.e., an atom that represents a reference to `X`. Conversely, the call `{Connection.take T}` returns the entity corresponding to the ticket `T`. Those functions allow a site to offer an entity reference to other sites via textual communication means. Indeed, as an atom is nothing more than a string of characters, it can be transmitted by e-mail, via a web site, or even told to the phone.

This mechanism is often used as a *bootstrapping* mechanism for distributing an application. The first entities that sites share are used to transmit other entities, by transitivity.

Managing resources. Not everything can be distributed. Assume a site b executes a procedure sent by site a, and that procedure has to save temporary data in a file. The site b may grant access to its file system, by it needs a way to provide this access to the running procedure. In order to solve this issue, Mozart has a module system based on *functors*. A functor is the specification of a module, with a list of modules to import, exported references, and code. If b receives a functor, it can instanciate it with a module manager that will provide (or possibly deny) the necessary local resources to the new module. Consult Mozart's documentation [Moz99] for more detail about functors.

3

Application structure and distribution behavior

With network transparency it is possible to take a program and distribute it, and it will run correctly. But it might be slow, for instance because its distribution involves much communication between sites. Here the programmer can take advantage of the network-aware aspects of the language to control the communication involved by the program's distribution. These aspects do not break transparency in the sense that the program is always a correct centralized program. So transparency gives two advantages. First, a centralized program is a correct distributed program. Second, tuning a centralized program for best distributed performance can be done by modifying the centralized program, e.g., with annotations that have no effect on centralized meaning. The program always retains a correct centralized semantics.

When tuning distribution, the fundamental distribution behavior is determined by the structure of the program. The latter defines the paradigms that are used: functional, dataflow, message passing, sequential or concurrent object-oriented, etc. The type of shared entities will determine communication patterns between sites. At a finer level, a given entity may be distributed in several ways, for instance stationary or replicated state, each having a specific distributed behavior. Choosing the most appropriate program structure is the way to make network transparency work.

3.1 Layered structure

We assume that a distributed application is structured in terms of *components*. We define a component as a program fragment with well-defined inputs and outputs. A component is itself defined in terms of simpler components. Exam-

ples of components are: a procedure, an object, several objects linked together. In this context, components can themselves be distributed. A distributed component can be decomposed into several components running on different sites, and communicating through shared language entities.

A component can be defined with a mixture of paradigms. Some of its subcomponent can be purely functional, while others use message passing and dataflow variables, for instance. The choice of paradigm is an advantage for reasoning about the program, since simpler components will require simpler reasoning techniques. For example, a declarative component handling lists will not be subject to race conditions, provided no other component concurrently binds its outputs.

The layered structure of the language encourages the programmer to pick the simplest programming paradigm to solve his or her problem. Concurrency with shared state is by far the most complex paradigm to program with. Most components do not need this expressive power. By limiting oneself to a part of the language, the programmer can take advantage of methodological support from the paradigm or the abstractions chosen. The network transparency ensures that this support is independent to whether the component is distributed.

The general advice is to keep the most general paradigm only for the components that need it, and to limit the extent of this paradigm inside the component itself. The usage of shared state concurrency is easier to manage when it is well confined in the program.

3.1.1 Using the declarative model

The simplest declarative components only provide stateless values, like pure functions without stateful dependencies. Dynamicity in declarative components are provided by shared logic variables. An example is a pipeline of components, where inputs and outputs are dataflow streams. Another example is several sites synchronizing on a given event. The sites only need to share one logic variable, and block until it is determined. The site notifying the event binds the variable to a conventional value, which automatically wakes up threads blocking on that variable.

Declarative components communicate by sharing values through logic variables. Communication is therefore purely dataflow and monotonic. From a distribution point of view, the programmer should pay attention to how data is shared among sites. Sharing stateless data is cheap in general, since it can be copied. But sharing too much data may imply much communication between sites.

Using the full power of the declarative model, one can also share lazy computations between components. What is shared is actually a logic variable. The by-need synchronization mechanism works through distribution.

Nondeterminism. Distributed declarative components are subject to nondeterminism in the sense that concurrent threads may bind variables in any

order. However, this nondeterminism is not *observable* from within the declarative model. If a declarative component terminates *without failing*, its outputs (defined by variable bindings) can be expressed as a mathematical function of its inputs. Their value do not depend on the execution order of the various threads in the component.

Note that if a failure occurs, for instance because of incompatible concurrent bindings, a part of the component will be failed state. From a strictly declarative point of view, the whole program has failed. But in the wider model, the failure does not automatically propagate to the whole program. If the rest of the program continues to run, then we have observable nondeterminism. But we are no longer in the declarative model.

3.1.2 Using message passing

When *observable* nondeterminism is required in a component, message passing is a good way to go. Ports provide a easy and efficient way to handle the nondeterminism in the component. The `Send` operation is asynchronous, and therefore it only requires a message to be sent by the system. The port's stream itself is monotonic, and behaves like a declarative component if it is distributed.

An effective use of this model is to let only one thread read the stream of messages, and treat them sequentially. This is the idea underlying *active objects*. An active object is a component that runs on only one site, and communicate with other active objects by sending messages.

Replying with variables. The model naturally offers two ways to reply to a given message. The simplest solution is to use the declarative model, and put a logic variable in the message. This logic variable will be bound to the reply. Snippet 3.1 on the following page shows two abstractions that implement this technique. The function `MakeServer` takes a function, and returns a port with a server. The server thread applies the function to each message, and binds the reply variable to the result of the call. The procedure `SendRecv` sends a message `X` to a port `P`, together with a reply variable `Y`. The code below creates a stateless server which adds 42 to each message it receives. Then the server is called with 54, with `Result` as the reply variable. As you can see, the functional notation allows to call `SendRecv` as a function.

```
Server={MakeServer fun {$ X} X+42 end}
Result={SendRecv Server 54}
```

Replying with continuations. A slightly more general technique is to put a *continuation* in the message, i.e., a procedure that is called by the receiver to reply the message. Note that the procedure is copied to the receiving site, so that it can be applied there. The continuation allows to program more sophisticated patterns of communication, like the *promise pipelining* provided in the language E [Mil06].

```
fun {MakeServer F}
   S in
   thread
      for X#Y in S do Y={F X} end
   end
   {NewPort S}
end

proc {SendRecv P X Y}
   {Send P X#Y}
end
```

Snippet 3.1: Abstractions to create and call a server with a reply variable

Snippet 3.2 on the next page defines a few abstractions that can be used to program in a "promise pipelining" style. The function `MakeServerC` creates a port with a server. The server thread applies a function `F` to each received message, and calls the continuation with the result. Let us create three servers `P`, `Q`, and `R` with that function. The server `P` replies either `Q` or `R`, depending on the message it receives. Servers `Q` and `R` expect an integer as message, and return the message after an arithmetic operation. Note that those servers should be created on different sites.

```
fun {F X} (if X==foo then Q else R end) end
P={MakeServerC F}
Q={MakeServerC fun {$ X} X+42 end}
R={MakeServerC fun {$ X} X div 2 end}
```

Now let us call `P`, thanks to the procedure `SendToPort`, with a continuation `C1` that is not determined yet. We will determine its value right after.

```
C1={SendToPort P foo}
%% is equivalent to: {Send P foo#C1}
```

The server will determine a result for the message, in this case `Q`, and eventually call `{C1 Q}`. During that time, the sender determines what `C1` does: it should send the message 54 to its argument. The fact that the continuation `C1` is called by server `P` makes that the message to `Q` is sent directly from `P`. There is no need to come back to the sender. The procedure `SendToCont` determines its first argument to do exactly that:

```
C2={SendToCont C1 54}
%% is equivalent to: proc {C1 Res} {Send Res 54#C2} end
```

The variable `C2` will thus be sent to `Q` as a continuation, so server `Q` will eventually call `{C2 96}`. We now define this continuation with the procedure `GetResultC`: `C2` binds its result to the variable `X`.

```
X={GetResultC C2}
%% is equivalent to: proc {C2 Res} X=Res end
```

```
fun {MakeServerC F}
   S in
   thread
      for X#Cont in S do {Cont {F X}} end
   end
   {NewPort S}
end

proc {SendToPort P Msg Cont}   % send Msg to port P
   {Send P Msg#Cont}
end
proc {SendToCont C Msg Cont}   % send Msg to promise C
   proc {C Res} {SendToPort Res Msg Cont} end
end
proc {GetResultC C X}          % get result from continuation C
   proc {C Res} X=Res end
end
```

Snippet 3.2: Promise pipelining with continuations

These continuations have defined the following pipeline: the client sends message `foo` with continuation `C1` to server `P`; then `P` sends message 54 with continuation `C2` to server `Q`; then `Q` sends its result 96 back to the client via the variable `X`. Note that this machinery relies on the fact that procedures are copied from site to site.

3.1.3 Using shared state concurrency

This is the most complex model from a programming point of view. And it is also the most demanding for the distribution. Shared state implies that read/write operations can be performed from multiple sites. Moreover, those operations are synchronous, and create many dynamic dependencies between sites. The difficulty stands in managing the state such that it is consistent: all the updates of a stateful entity must be serializable, as in a centralized multithreaded program.

Moreover, the shared state concurrency model is very sensitive to errors. This is because the threads follow an interleaving semantics. Consider the following example, where two threads perform each a read and a write operation on the cell `C`.

```
C={NewCell 0}
thread C:=@C+1 end     % thread A
thread C:=@C+2 end     % thread B
```

The result of the execution does not depend on where the threads and the cell are localized in the distributed system. When both threads terminate, the cell may contain either 1, 2, or 3. Some executions lead to surprizing behaviors. For

instance, the cell contents may decrease in this execution: thread A reads 0, then thread B reads 0, then thread B writes 2, then thread A writes 1.

Managing the nondeterminism in the programming language can be a problem. Using locks can help to create critical sections into the code, but they quickly lead to deadlock avoidance issues. In both the centralized and distributed cases, the programmer should use *transactions* to handle atomicity issues in his or her program. Transactions can be implemented quite efficiently in a centralized setting [ST95, VH04].

Implementing a distributed transaction system with good network behavior is not an easy task. Two systems have been proposed so far by our research team. The "GlobalStore" is fault-tolerant transactional replicated object store designed and implemented by Iliès Alouini and Mostafa Al-Metwally [AM03]. It takes advantage of replication to reduce latency when computing the transaction. Another transactional system was proposed to run on structured overlay networks, and was designed by the author and Valentin Mesaros [MCGV05]. The latter uses transaction priorities to avoid deadlocks, and the two-phase commit algorithm to ensure consistency between sites. Note that these systems only handle permanent failures.

3.2 Classification of language entities

The design of Oz is such that different entities may use different distribution protocols, and thus have different network behaviors. In fact, the distributed behavior of the whole program is determined by what type of entities are used, and how they are shared among sites. This section explores the different kinds of entities in Oz, and how they can be distributed.

Entities can be partitioned into three main categories: mutable, immutable, and monotonic. Each category has specific requirements that influences their possible distributed behavior. All the entities in a given category share the same set of distribution protocols. So the category of an entity determines what possible protocols it may use, and thus what possible network behavior it may have. We present each category, with the protocols available for each, and examples of Oz entities in those categories.

3.2.1 Mutable entities

This is the category of stateful entities in general. Those entities have an internal state, and the distribution must maintain a globally consistent view of the state. Here we sketch three possible distribution strategies for them.

- The simplest way to distribute a mutable entity is to make its state stationary. All operations are sent to the site holding the state, performed there, and a value can be returned.

- In the "mobile state" strategy, the state moves on the sites where operations are attempted. When the state is on a given site, it can be accessed locally. The state behaves like a cache, since several operations can be performed locally before the state leave the site.
- One can also replicate the state through sites. An update of the entity first invalidates all copies, then sends the new state. Reading the state is a pure local operation, and it can be done immediately if the copy is valid. This scheme is efficient when reading the state is more common than updating it.

Read and write operations are synchronous in general. Oz cells, objects, locks, dictionaries, threads belong to that category. Ports can be considered a special subcategory: the operation `Send` is an asynchronous update. The simplest strategy in this case is to leave the state stationary. To make an update it is enough to send a message to the site holding the state.

3.2.2 Monotonic entities

Those are also stateful, but their state is updated in a monotonic way. From a distribution point of view, they are more flexible that mutable entities. Their state can be replicated without the need to synchronize all the sites to perform an update. Single-assignment variables and streams are examples of monotonic entities.

Monotonic entities support the concurrent constraint operations *ask* and *tell* [Sar93]. The ask operation is just like a read. The tell operation updates the state of the entity. To ensure the consistency of the tell, all updates are serialized on one site. This site forwards the operation to all the other sites.

Single-assignment variables are *transients*, which is a subcategory of monotonic entities. Transients have a final state where they become another entity. The entity exists until is is bound. In Oz, logic variables have three states: free and not needed, free and needed, and determined. Note that streams are built from transient entities (read-only views), and therefore inherit from the distributed behavior of transients.

3.2.3 Immutable entities

Those are constants. One can only read them. They are usually copied eagerly or lazily between sites. When the entity has an identity, the protocol can guarantee that the copy is done once. This is useful when the value is large. In some cases the value cannot be copied, for instance because of implementation or security limitations. Read operations are then performed like in the mutable case.

Immutable entities range from simple values (atoms, numbers), to compound values (records), and even closures (procedures, classes). A compound value is copied with its fields, which may also be compound values or closures.

The copy of a compound value should have the same structure as its origin, including possible cyclic references and coreferences. Cycles and coreferences are detected when the value is serialized, so that they are not an issue for the programmer.

Closures are extremely powerful, because they contain both code and external references. The latter are handled just like records. And the code is copied between sites. The promise pipelining example in Snippet 3.2 on page 31 makes use of them, for instance. Note that cycles may also happen with closures, since closures can contain record references, which can contain references to closures. An again, the cycles are gracefully handled by the system, such that each entity in the reference graph of a given entity is copied at most once.

3.3 Annotations

The application is structured into components, and those components are themselves decomposed, layer by layer, up to primitive components, i.e., language entities. In its first implementation of distribution, Mozart was providing its distributed entities with fixed behavior for each. It was considered that those choices were expressive enough for the programmer to code the distributed behavior of his or her choice. Objects were distributed with mobile state. Stationary objects had to be reimplemented in Oz on top of ports, for instance.

We now let the programmer choose the distributed strategy for each language entity in his or her program. This choice is stated by *annotating* the entity. An application can be structured from top to bottom, with the annotation system providing the shaping of the distributed behavior at the lowest level of the structure. Annotations are part of the *network awareness* of the language, since they give some explicit control on an entity's distributed behavior.

Annotations may cover many facets of the distribution system. The first and most evident one is how the state of an entity is distributed, and the impact on primitive operations on that entity. Another facet is the distributed memory management of that entity. The system could also provide some robustness for its entities, and annotations may help to parameterize how robust an entity must be.

How to annotate. Conceptually an annotation is a bit like a declarative statement. It states something about an entity. It can even be thought as a constraint that the user posts about the distributed behavior of an entity. It is not fully declarative, since logic variables have a specific status. Annotating a variable is not equivalent to annotating its value.

In our proposal, which is explained in detail in the next section, annotating an entity is done by the statement

`{Annotate` *entity parameter*`}`

The nice property about annotations is that they can be ignored in case of no failure. They do not change the semantics of the program in that case. Moreover, if the program is not distributed at all, they will not be taken into account by the system.

3.3.1 Annotations and semantics

Annotations describe programmer choices for the distribution of an entity. Network transparency implies that the distribution must be an implementation, or a refinement, of the entity's semantics. The centralized semantics of an entity often admits several distributed semantics. Annotations give the possibility to the programmer to specify which distributed semantics should be used for a given entity. The semantics of the language is given in chapters 6 and 7. The latter also gives the semantics of annotations themselves.

The semantics of a language entity is thus partly reflected in the application. The annotation system let the latter make semantic choices for language entities *at runtime*. We could say that annotations allows the programmer to change the semantics of the language. But one can only *choose* between semantic variants, which are well defined, and do not break the centralized semantics of an entity in case of no failure.

Annotations are thus a limited form of *reflection* in the programming language. Its boundaries are defined by the programming system and by the centralized semantics. The programmer may tweak an entity's semantics within safe boundaries.

3.3.2 Annotation system

In practice an annotation specifies parameters of the distribution subsystem. The actual annotation system goes slightly beyond the conceptual level, because it allows implementation compromises that may break the semantics of an entity. The typical example is the time-lease based garbage collector, which may remove an entity from memory even if some sites in the application still refer to it.

The procedure `Annotate` is called to specify distribution parameters for an entity. We have chosen annotations to be *monotonic*: you cannot change your mind once you have chosen an option. Moreover, once an entity is actually distributed, i.e., when it has been shared by at least two sites, its distribution parameters can no longer be changed. As a consequence, an entity can only be annotated *before* it gets distributed. It is therefore useful to annotate entities into the abstractions that create them.

Distribution parameters are specified as atoms or records. For instance, stationary state is specified with the atom `stationary`, and the use of a mobile access reference is specified by the record `access(migratory)`. Several parameters are combined in a list, like in the example

```
{Annotate E [stationary access(migratory) lease]}
```

We will see in the next section that this statement is equivalent to the following three ones.

```
{Annotate E stationary}
{Annotate E access(migratory)}
{Annotate E lease}
```

3.3.3 Partial and default annotations

Our annotation system is not only monotonic, it is even incremental. Several annotations can be put on a given entity, at different times. The result is that the entity is annotated by the conjunction of them, provided that it is consistent. For instance, mobile and stationary state are inconsistent together, but stationary state and time-lease garbage collection are consistent, because both parameters are orthogonal to each other. In our implementation, Mozart considers three orthogonal distribution parameters, namely the access architecture, the state protocol, and the distributed garbage collection. Those are described in more detail in the next sections.

The system also permits *partial* annotations. It means that some distribution parameters may be left unspecified whenever an entity becomes distributed. In that case, the system *completes* the annotation with default values. For instance, if the programmer annotates a cell with `lease` (time-lease based garbage collection), the system may complete the annotation to

```
[migratory access(stationary) lease]
```

right before distributing the cell.

Each type of entity has a default annotation, giving a value for each distribution parameter. The default annotation must be complete, of course. The system implementation may or may not allow the programmer to modify default annotations. In our prototype, default annotations can be modified by the program at any time.

3.3.4 Access architecture

The access architecture of an entity defines how all the sites sharing the entity coordinate with each other. This architecture is the base of the other protocols (state and reference consistency, see below). It could be anything, as long as one can implement the entity operations and garbage collection on top of it.

In Mozart, all sites sharing the entity own a *proxy*, and all those proxies refer to a unique *coordinator*, which is hosted by one of the sites. It is used by the other protocols as a reference point. When an entity reference is sent from one site to another, a network address of its coordinator is given. This allows the receiver to connect its proxy to the other proxies of the same entity in the whole system. The architecture is similar to a client-server architecture.

Knowing the type of access architecture already gives some information about the network behavior of the entity. For instance, an entity with stationary state will have its state located on the same site as the entity's coordinator. Another example is garbage collection, where the coordinator determines whether the entity is referred to by remote sites. If not, the entity is no longer distributed.

Mozart/DSS defines one parameter for the access architecture, which states whether the coordinator is stationary or mobile.

- `access(stationary)`: the coordinator is located where the entity was created, and remains on that site. This is the simplest strategy to manage the access architecture.
- `access(migratory)`: the coordinator can be moved from one site to another by a specific operation. There are several possibilities on how proxies can find where it is. Details can be found [Kli05].

A single point of failure. The coordinator of an entity is obviously a single point of failure. When it crashes, the entity's proxies are usually no longer capable of finding each other. Note however that we have a single point of failure *per entity*. This design decision is motivated by the fact that entity protocols should not solve all the problems. Entities are generally not robust to failures. Instead, failures are detected, and can be handled at the language level. Fault-tolerant protocols can be implemented in Oz, and hide failures by abstractions.

3.3.5 State consistency protocols

Those protocols implement the operations of the entity itself. The choice of protocol depends on the entity's nature, i.e., mutable, immutable, or monotonic. This parameter is the most important when considering the network behavior of entity operations. Here are the protocol annotations considered in Mozart/DSS.

1. Mutable entities.
   - `stationary`: the state of the entity is located on the site of the entity's coordinator. All operations on the entity's state are performed on this site. Synchronous operations therefore need a full round-trip from the requesting site to the coordinator to complete.
   - `migratory` and `pilgrim`: the state of the entity migrates from one site to another, and a site executes operations locally when the state is on that site. The state behaves like a cache: once the state is on a site, that site may perform several operations without extra network overhead.

- `replicated`: the state of the entity is copied on all the sites that use the entity, and the copies are synchronized by a two-phase commit protocol. This protocol is useful for data structures that are rarely updated. Read operations can be performed locally, while write operations require an atomic update of all sites using the entity.

2. Monotonic entities.

   - `variable`: this corresponds to the protocol described in [HVB+99]. The binding of the variable is performed on the site of the entity's coordinator.
   - `reply`: this is a variant of the `variable` protocol, where the binding is done on the first site that receives the reference. This protocol has the best network behavior when the receiver site attempts to bind the variable. The variable is typically used to a reply to a query.

3. Immutable entities.

   - `immediate`: the value is sent together with the reference of the entity. The unicity of the entity is guaranteed, even if its value is sent multiple times. All values with structural equality (numbers, atoms, records) use this protocol.
   - `eager` and `lazy`: those protocols guarantee that the value is sent at most once. When the entity is sent to a site, only its reference is actually sent. The receiving site requests the entity's value if it does not have it yet. In the `eager` case, the value is requested upon receipt, while in the `lazy` case, it is requested once the value is actually needed.
   - `stationary`: the value is not copied on other sites. Remote operations require a full round-trip to the coordinator. For instance, one can provide access to a chunk without allowing copies on possibly untrusted sites.

3.3.6 Reference consistency protocols

Those protocols ensure the reference integrity, by implementing a distributed garbage collector. Here the choice is not exclusive: one can combine several protocols in a single annotation. An entity is kept in memory if all protocols require so. Mozart proposes three algorithms:

- `persistent` simply keeps the entity alive forever on its coordinator. The entity is simply never removed by the garbage collector. This can be useful for providing a service on a site, which should run until the site terminates.

- `refcount` uses a weighted reference counting scheme. Each reference to the entity is assigned a weight, and the entity's coordinator keeps track of the sum of the weights of all remote references. When this number reaches zero, the entity is no longer distributed. The advantage of using weighted references is that new remote references can be created without notifying the coordinator. It suffices to keep the total weight constant.

- `lease` uses time-lease based mechanism. Sites holding a remote reference to an entity regularly notify its presence to the coordinator of the entity. The time between successive notifications is called the lease period. If the coordinator has not been notified during a long time (typically much longer than the lease), the entity is no longer considered distributed.

The algorithm `refcount` guarantees consistency, i.e., a coordinator remains alive while its proxies are, even in case of long network failures, but is not robust to site failures. The algorithm `lease` does not guarantee consistency in case of network delays, but handles site failures gracefully. It is up to the programmer to choose what fits best for his or her application.

3.4 Related work

How do other systems provide tuning of a distributed program? Are they easy to tune at all? Can the tuning process be programmed in the language, or is it external to it? In the latter case, are the tuning techniques heavily modifying the program?

3.4.1 Erlang

In the Erlang philosophy, everything is a *process*, and the only communication primitive between processes is message passing with values. Processes are independent of each other, and cannot share memory. Every process is sequential and programmed in functional style. This simple model fits pretty well with distribution, and allows efficient implementations.

Process identifiers can be sent between sites, and sending messages to a remote process is transparent. Processes do not migrate between sites, and garbage collection is up to the programmer. The language favors lightweight client-server style. For instance, the Open Telecom Platform (OTP) provides generic server abstractions, which support transactional semantics (crashed servers are restarted with a valid former state) and code swapping (the server code can be changed on-the-fly). In fact, the OTP provides many more abstractions to build large-scale, fault-tolerant, distributed applications [Arm07].

With the simplicity of Erlang's programming model, any communication pattern can be programmed with processes and messages. Libraries like the OTP already provide powerful abstractions for distributing applications. Of course, the programmer should always use this kind of abstraction to build

his or her applications. Changing the network behavior can then be done in a modular way. Using directly the distribution facilities makes the program harder to adapt.

3.4.2 Java RMI

There is no distribution mechanism defined by the language Java itself. But the Java Remote Method Invocation (RMI) library has quickly become the most popular distribution mechanism in the Java community. The library provides two ways to distribute an entity: full serialization and remote method invocation. In the first case, a copy of the object is made once a reference to that object is sent to a site. In the second case, only a reference to the original object is created on the receiving site. When that reference is invoked, the method invocation is sent to the object's site and the calling thread waits for its termination. These objects are said to be *remote*, while fully serialized objects are *non-remote*.

This mechanism imposes synchronous interaction between sites, which can be slow in a distributed setting. But worse, reference integrity is only guaranteed per method invocation, and not in general [Sun97]. The consequence is important: distributed objects have a *different semantics* than centralized objects.

Besides these semantic issues, Java RMI is not transparent to the programmer. Remote objects must implement the interface `java.rmi.Remote`, while non-remote objects implement `java.io.Serializable`. Turning a local object in a distributed one requires to modify its class. The library provides a few abstractions to write servers, though.

3.4.3 E

The language E is an object-oriented programming language designed for secure distributed computing. It was created by Mark S. Miller, Dan Bornstein, and others at Electric Communities in 1997. It combines capabilities and a message passing concurrency model with Java-like syntax. Its concurrency model is based on event loops and promises, in order to prevent deadlocks. More on security aspects of E can be found in Mark Miller's thesis [Mil06].

Objects behave like concurrent sequential agents with synchronous or asynchronous method invocation. Each object is stored into a *vat*, which is the unit of localization. Synchronous invocation can only happen between objects in the same vat, and corresponds to a sequential method call.

```
def result := bob.foo(carol)     /* synchronous call */
println(‘done: $result‘)
```

A method is always executed atomically, and should never block or run forever. This strong limitation to concurrency was chosen to avoid common programming errors due to shared state concurrency.

Objects can also invoke each other asynchronously, with the *eventually* or *send* operator `<-` (see below). If the method returns a result, the operation returns immediately a promise for the result.

```
def result := bob <- foo(carol)     /* eventual send */
when (result) -> {                  /* promise resolution */
   println(‘done: $result‘)
} catch problem {
   println(‘oops: $problem‘)
}
```

One can send to the promise immediately; a promise pipelining mechanism ensures that the resulting object is eventually sent the method (like in Section 3.1.2 on page 29). One can also synchronize on the result with a `when` statement. Once the promise is resolved, the code is eventually run (atomically). The `catch` part allows to handle a promise failure.

The distributed model of E is strongly determined by the security aspect of the language. By default, vats do not trust each other. Therefore objects are never copied or moved between vats. Vats are strongly isolated from each other, and inter-vat communications are encrypted. The encryption also guarantees that object references cannot leak into intermediate vats in the promise pipelining process. The distribution model is thus limited to client-server communication with promises, but the capability system is guaranteed safe.

4

Asynchronous failure handling

We go one step further in the distribution support by reflecting partial failures in the programming language. We propose a language-level fault model that is compatible with network transparency. Because a site or network failure may affect the proper functioning of a distributed entity, our model defines how entities can fail, and how those failures are reflected at the entity level in the language. Here are the principles of the model, each being described in the corresponding subsection below.

- Each site assigns a *local fault state* to each entity, which reflects the site's knowledge about the entity.
- There is no synchronous failure handler. A thread attempting to use a failed entity blocks until the failure possibly goes away. In particular, no exception is raised because of the failure.
- Each site provides a *fault stream* for each entity, which reifies the history of fault states of that entity. Asynchronous failure handlers are programmed with this stream.
- Some fault states can be enforced by the user. In particular, a program may explicitly provoke the global failure of an entity.

This fault model is an evolution of the first fault model of Oz, and integrates parts of another proposal made by Donatien Grolaux *et al.* [GGV04]. The next chapter will demonstrate that it improves the ease of programming and modularity of failure handlers. A comparison with the other fault models of Oz is given in Section 4.5 on page 56.

4.1 Fault model

We first provide a precise description of the kind of faults we consider in the system in which the program runs. This system is composed of sites that communicate through a network. We model a certain number of failures in that system, how they affect the system, and how they can be detected. Failures affect language entities, so we also consider failures at the entity level. The model we use here is inspired from Rachid Guerraoui *et al.* [GR06], and is quite standard in the field of distributed systems.

Note that we will sometimes use the term *process*. A process is simply something that has some autonomous behavior, some internal state, and which interacts with other processes by exchanging messages through the network. Both sites and language entities can be considered as processes.

4.1.1 Failures

Site failures. A site may fail by *crashing*, i.e., at a given time t, it stops doing anything, especially sending and receiving network messages. There is no recovery mechanism by default, the failure is permanent. This kind of failure is called *crash-stop* or *fail-stop*. A site that has not crashed yet is said to be *correct*.

We assume that sites are subject to neither *omission* (where the process may "miss" some messages), nor *Byzantine faults* (where the site may perform any arbitrary action). Those are much harder to handle, and detect. The lack of generic detection mechanism makes any kind of language support for them virtually impossible.

Network failures. The network is considered *reliable* in the sense that a message sent by a site a to a site b will eventually be delivered, unless a or b crashes. Messages are never corrupted nor delivered more than once. However, the communication between two sites may take arbitrary time. The communication link may appear to have failed if no message is delivered to the destination site during a long but finite period of time. In other words, network failures can be defined as communication delays that are longer than expected. We do not consider the case where network failures would be permanent. Such a failure would mean that a site can no longer communicate with any other site. By convention, network failures are always considered to be temporary.

Entity failures. Language entities are subject to the same kind of failures as sites. A failed entity stops being functional. No language operation can have an effect on it. Its state is lost, and there is no recovery mechanism. The failure is permanent and global: a failed entity is unusable for all sites.

We also consider a failure of type fail-stop, but which is valid for a given site: the entity is crashed for that site, but may be functional for other sites.

In other words, that site can no longer use the entity. We say that the entity is *locally failed* on the given site. This failure is triggered by the operation `Break`, which is described in Section 4.3.2. It is typically used to prevent the site from using the entity, and does not affect the other sites.

4.1.2 Failure detectors

In order to handle failures at the program level, we need *failure detectors*. A failure detector is a component that tries to determine whether a given process has crashed. It notifies the program when it *suspects* the process to have crashed. Not all failure detectors are identical, there exists several types of them. We can classify them according to three properties: their completeness, accuracy, and monotonicity.

- A failure detector is *complete* if a crashed process is always eventually suspected.
- A failure detector is *accurate* if a suspect process has actually crashed, and *inaccurate* if it may suspect a correct process. It is *eventually accurate* if no correct process is suspected forever.
- A failure detector is *monotonic* if a suspect process is never notified correct later. In other words, the failure detector never "changes its mind."

The completeness is a liveness property, it ensures the eventual detection of a crash. On the other hand, the accuracy is a safety: it prevents erroneous suspicions. Those two properties are essential for reasoning about the correctness of an algorithm that handle failures. The monotonicity also helps for reasoning, because monotonic detectors have a simpler behavior than nonmonotonic ones.

A failure detector is *perfect* if it is complete and accurate. It is *eventually perfect* if it is complete and eventually accurate. Perfect failure detectors are not so common, because they require strong properties of the underlying system. For instance, perfect detectors are possible on a local area network (LAN), but not on the internet in general. For the internet, one has to use eventually perfect detectors.

Three simple failure detectors. Our model proposes a combination of three failure detectors, namely *tempFail*, *permFail*, and *localFail*. Each detector has its own properties in terms of completeness, accuracy, and monotonicity. We will show in the next sections how we use them to handle language entity failures.

- The *tempFail* detector is eventually perfect. It uses two notifications: `tempFail` and `ok`. The first one occurs when the target process is suspected, the second one occurs when the detector has found evidence of correctness of the process. This detector is nonmonotonic.

detector	complete	accurate	monotonic
tempFail	yes	eventually	no
localFail	no	no	yes
permFail	no	yes	yes

Table 4.1: Summary of the properties of the three failure detectors (for global failures)

- The *permFail* detector is accurate but incomplete. It is not guaranteed to detect a crash, but it never erroneously reports a crash. It uses the notification `permFail`. By definition it is monotonic.
- The *localFail* detector is perfect for local failures, and uses the notification `localFail`. However it is neither complete, nor accurate for global failures in general. Its completeness and accuracy depend entirely on the program. However it is monotonic: suspicion remains forever.

A summary of the properties of the three failure detectors is shown in Table 4.1. Note that these properties are given with respect to global failures. This is why *localFail* is neither complete nor accurate.

4.1.3 Entity fault states

For each entity e, every site has a failure detector that combines the three failure detectors *tempFail*, *permFail*, and *localFail*. That failure detector maintains a *local fault state*, which is like a *view* of the actual fault state of the entity. The failure detector sends a notification at every state transition. The notification mechanism is described in Section 4.2.3.

The failure detector has four states, called *local fault states*, or *fault views*: `ok`, `tempFail`, `localFail`, and `permFail`. Valid state transitions are depicted in Figure 4.1 on the facing page. The semantics of the states are the following.

- `ok` is the initial state, and can also be triggered by the *tempFail* detector. It means that the entity is not suspected by any of the basic failure detectors.
- `tempFail` is triggered by the *tempFail* detector. It means that the site is temporarily unable to complete any operation on the entity. This typically happens when this site cannot communicate with other sites that are necessary for performing language operations on the entity.
- `localFail` is triggered by the *localFail* detector. It means that the entity is permanently unavailable for this site. Note that it is local, i.e., other sites may still have access to the entity. This state can be enforced by the program.

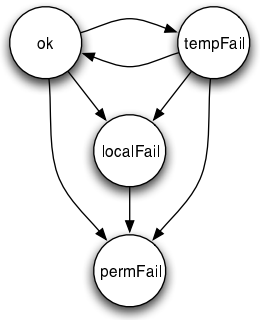


Figure 4.1: Local fault state diagram of an entity

- `permFail` is triggered by the *permFail* detector. It means that the entity has crashed. No site can ever perform an operation on it. This state is final.

The main advantage of this model is that it provides a simple yet precise description of an entity's state, from the viewpoint of one site. It abstracts the type of hardware and system used, the protocols, and even the kind of entity it applies to. It describes the failure from the programming language's point of view. Yet its simplicity still allows to reason about the partial failures in a program.

4.1.4 Concrete interpretation of fault states

Knowing the kind of an entity and its distribution strategy, one can easily give a more precise interpretation of a fault state. Here we give the various concrete reasons for an entity to fail. The only concepts we rely on are the ones given in Chapter 3. All failures can be expressed in terms of sites, protocols, coordinators, and memory management.

Note that the interpretation we provide for fault states is of course related to how the distribution of an entity is implemented. A sophisticated implementation of distribution would have led to a complex fault model. In our work we favor a simple fault model, therefore keeping the implementation simple. The programmer should be able to reason easily about the properties of the distribution. Complex fault-tolerant abstractions should be built at the higher user level, not at the low level.

Mutable entities. Two sites are usually involved when reasoning about mutable entity failures: the coordinator site and the site holding the state. The coordinator is necessary in all protocols to manage the entity's state. It is the site holder if the state is stationary; it manages to bring the state to the requester in case of a mobile state; and it ensures mutual exclusion when the

state is replicated. The state holder is also crucial: its failure always implies that the state is lost.

A mutable entity is in state `tempFail` if the coordinator or state holder is unreachable. The state `localFail` is triggered by the program. The state `permFail` is reached when the coordinator or the site holding the state has crashed, or the coordinator has removed the entity from its memory. The coordinator crash can be provoked by the program (see Section 4.3).

The second reason for the state `permFail` has already been mentioned in Chapter 3: time-lease based garbage collection is not correct in case of network failures. The coordinator considers that the entity is no longer used when no other site has showed interest for a certain duration. A problem arises when a site cannot reach the coordinator because of a network problem. In that case, the entity will fail on that site. The good property is that it is diagnosed properly, and reflected in the language. If the network recovers and the site can reach the coordinator again, the removal of the entity will be notified, and result in the entity failure.

Monotonic entities. Transients are pretty similar to mutable entities when it comes to failures. In the protocol `variable`, the coordinator is also a state holder. The state holder might be different in the `reply` protocol. If only one site refers to the variable besides the coordinator, then that site is the state holder. The same reasoning as with mutable entities applies here.

A property of logic variables is that they conceptually disappear once they are bound. In fact, bound variables have reached their final state, and become invisible to the program. For the sake of consistency, bound variables do not fail, and failed variables remain unbound.

Immutable entities. Immutable entities are simply values. Their possible fault state depend on whether they are copied between sites (protocols `immediate`, `eager`, `lazy`) or not (protocol `stationary`). Note that entities using the `immediate` protocol never fail, since one cannot have a reference to the entity without having its state. As all entities with structural equality (numbers, atoms, records) use this protocol, they are not subject to failure.

Values cannot fail permanently if they are copied between sites. If the site from where the copy is made is unreachable or has crashed, a temporary failure will be notified. The local fault state can even be `localFail`. But the state `permFail` should never be observed, because any other site may provide a copy of the value. The fault state `permFail` requires that no copy of the value is available anywhere, even in a file. This property is difficult to verify in practice.

Values distributed with a stationary state are different. This protocol can be used when copying the whole value is too costly or insecure. Remote sites can still access the value, typically with the dot operation "`.`". In that case, the causes of failure are the same as mutable entities with stationary state.

4.2 Failure handlers

We now discuss the possible ways to handle entity failures in the language. We make a clear distinction between two basic ways of handling failures, namely *synchronous* and *asynchronous* handlers. As we shall see, asynchronous failure handling is preferable to synchronous failure handling.

4.2.1 Definition

A *synchronous* failure handler is executed in place of a statement that attempts to perform an operation on a failed entity. In other words, the failure handling of an entity is synchronized with the use of that entity in the program. Raising an exception is one possibility: the failure handler simply raises an exception. In contrast, an *asynchronous* failure handler is triggered by a change in the fault state of the entity. The handler is executed in its own thread. One could call it a "failure listener". It is up to the programmer to synchronize with the rest of the program, if that is required.

The following rules give small step semantics for both kinds of handlers. The symbol σ represents the store, i.e., the memory of the program. The store is partitioned among the sites, and the elements of the store that are specific to a site a are subscripted by a. Each site a reflects its view of the fault state of an entity in the store through a system-defined function $\mathsf{fstate}_a(x)$, which gives the local fault state of x. Each execution rule shows on its left side a statement and the store before execution, and on the right side the result of one execution step.

Rule (4.1) describes the semantics of a synchronous failure handler. It states that a statement S attempting an operation on entity x can be replaced by a handler H if the fault state of entity x is not ok, i.e., if x has failed.

$$\frac{S_a \parallel H_a}{\sigma \parallel \sigma} \quad \text{if} \quad \begin{array}{l}\text{statement } S \text{ uses entity } x \\ \text{and } \sigma \models \mathsf{fstate}_a(x) \neq \mathsf{ok}\end{array} \tag{4.1}$$

Rule (4.2) gives the semantics of an asynchronous failure handler. A new thread is spawned with handler H whenever the fault state of x changes. Note that there may be more than one handler on x; we assume all handlers are run when the fault state changes.

$$\frac{}{\sigma \wedge \mathsf{fstate}_a(x){=}fs \parallel \sigma \wedge \mathsf{fstate}_a(x){=}fs'} \quad \begin{array}{c}H_a \\ \text{if} \quad fs \rightarrow fs' \text{ is valid}\end{array} \tag{4.2}$$

4.2.2 No synchronous handlers for Oz

In Oz, when the fault state of a given entity is not ok, operations on that entity may not succeed. Raising an exception in that case might look reasonable, but our experience suggested that it is not. Because of the highly concurrent nature of the language, raising exceptions quickly creates race conditions between

threads. The functional code is cluttered with failure handling code. Other kinds of handlers have been tried, but without success.

We have chosen to use asynchronous failure handlers only. We propose the following model.

> "Failure causes blocking": an operation on a failed entity simply *blocks* until the entity's fault state becomes `ok` again.

The operation naturally resumes if the failure proves to be temporary. It suspends forever if the failure is permanent (`localFail` or `permFail`). With this model, nothing extra can happen in a program that does not handle distribution failures.

4.2.3 Entity fault stream

In our proposal, asynchronous failure handlers are programmed as threads that monitor entities, and take action when an entity changes its local fault state. On every site, each entity is associated with a *fault stream*, which reflects the history of the fault state's view of the entity. The system maintains the *current* fault stream, which is a list $fs|s$, where fs is the current view of the fault state, and s is an unbound variable. It is defined semantically as a system-defined function $\mathsf{fstream}_a(x)$ that returns the current fault stream of the entity x on site a. The semantic rule

$$\frac{}{\sigma \wedge \mathsf{fstream}_a(x){=}fs|s \;\Big\|\; \sigma \wedge \mathsf{fstream}_a(x){=}s \wedge s{=}fs'|s'} \quad \text{if} \quad fs \to fs' \text{ is valid} \tag{4.3}$$

reflects how the system updates the fault state to fs'. The dataflow synchronization mechanism wakes up every thread blocked on s, which is bound to $fs'|s'$. An asynchronous handler can thus observe the new fault state simply by reading the elements of the fault stream.

To get access to the fault stream of an entity x, a thread simply calls the function `GetFaultStream` with x, which returns the fault stream of x on the current site. A formal definition is given below. To read the current fault state, one simply takes the first element of the returned list.

$$\frac{(y{=}\texttt{\{GetFaultStream}\ x\texttt{\}})_a}{\sigma} \;\Big\|\; \frac{(y{=}fs|s)_a}{\sigma} \quad \text{if} \quad \sigma \models \mathsf{fstream}_a(x){=}fs|s \tag{4.4}$$

Figure 4.2 on the next page shows an example of how an entity's fault stream may evolve over time. The stream is a partially known list, and the underscore "_" denotes an anonymous logic variable. In the last step, the stream is closed with `nil`. This may happen in two situations, which are explained below. Snippet 4.1 on the facing page shows a thread monitoring an entity `E`, and printing a message for each fault state appearing on the stream. The printed message is chosen by pattern matching. The thread is woken up each time the stream is extended with a new state.

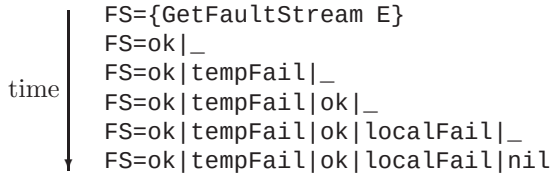


Figure 4.2: An example of a fault stream evolving over time

```
thread
   for S in {GetFaultStream E} do
      T = case S                          % pattern matching on S
          of ok        then "entity is fine"
          [] tempFail  then "some problem, don´t know"
          [] localFail then "no longer usable locally"
          [] permFail  then "no longer usable globally"
          end
   in
      {Show T}
   end
end
```

Snippet 4.1: A thread that prints messages when entity `E`'s fault state changes

Special case: variables. Monitoring variables requires a bit more care than other entities. This is because variables conceptually disappear once they are bound: they *become* what they are bound to. The question is: what happens to the fault stream of a variable once the latter is bound? There are two distinct cases to consider, as the variable is bound to either a value, or another variable.

Consider a variable x that is bound to another variable y. From a programmer's point of view, the binding is transparent: x remains a variable. For a thread monitoring x, it seems quite natural to smoothly switch to monitoring y. We propose to make this transition automatic by *merging* the fault streams of x and y. Basically the tail of the fault stream of x is bound to the tail of fault stream of y, prepended by y's current fault state if it is different from x's current fault state. This binding makes sure that the monitor does not miss a fault state.

The other case we have to consider is the binding of the variable to a value. We think that merging the fault streams is not a good idea here, because the entities are of different nature, and this may lead to confusion. However, we need a clear mechanism to notify the monitoring thread that the variable has been bound. We propose to *close* the fault stream by binding its tail to `nil`, because the variable has conceptually disappeared. Once this happens, calling `GetFaultStream` on the variable will return the fault stream of its value.

Failure history. The fault stream of an entity e on a site a reifies the history of fault state observations of e by a. Moreover it transforms the nonmonotonic changes of a fault state into monotonic changes in a stream. It provides an almost declarative interface to the fault state maintained by the system. This interface looks much simpler and more elegant than registered handlers, which is what Mozart used before [VHB99]. In particular, the fault stream guarantees that the failure handler cannot miss a state transition.

Note that the fault stream may also be closed, i.e., its tail bound to `nil`, whenever it is no longer maintained by the system. This is performed by the system when the entity is no longer in memory. See Section 4.4.

4.2.4 Discussion

Synchronous failure handlers are natural in single-threaded programs because they follow the structure of the program. Exceptions are handy in this case because the failures can be handled at the right level of abstraction. But the failure modes can become very complex in a highly concurrent application. Such applications are common in Oz and they are becoming more common in other languages as well. Because of the various kinds of entities and distribution protocols, there are many more interaction patterns than the usual client-server scheme. Handlers for a given entity may run in many threads at the same time, and those threads must be coordinated to recover from the failure.

All this conspires to make fault tolerance complicated to program if based on synchronous failure handling. This mechanism was in fact never used by Oz programmers developing robust distributed applications [GGV04]. Instead, programmers relied on the asynchronous handler mechanism to implement fault-tolerant abstractions. One such abstraction is the "GlobalStore", a fault-tolerant transactional replicated object store designed and implemented by Iliès Alouini and Mostafa Al-Metwally [AM03].

4.3 Making entities fail

Failures in distributed systems are often partial. This will be the case with entity failures in a distributed application, especially if the programmer defines components that are spread among many sites. In many cases, if a subset of the entities of a component have failed, the component itself might no longer function. The components that use the failed component must be able to detect the failure, and trigger a recovery mechanism. The question is: which entity should they monitor?

A component should not monitor all entities of another component explicitly. This would prevent any encapsulation in the monitored component. But monitoring the entities it has access to might not be enough, if none of the monitored entities fails. One possibility is to design a component-level protocol that makes sites consider the component as failed. Another possibility is to

```
proc {SyncFail Es}
   Trigger in
   for E in Es do
      thread
         if {List.member permFail {GetFaultStream E}} then
            Trigger=unit
         end
      end
   end
   thread
      {Wait Trigger}
      for E in Es do {Kill E} end
   end
end
```

Snippet 4.2: Synchronize the failure of a set of entities

make the monitored entities fail on purpose. We propose to provide support for the second alternative in our failure model, i.e., the program can make an entity fail.

4.3.1 Global failure

We provide a new operation to make an entity fail. The statement `{Kill` e`}` attempts to make the entity e permanently failed, i.e., in fault state `permFail`. The operation is asynchronous, which means that it returns immediately, and is idempotent. It initiates a protocol that tries to make the entity globally failed. Once it is done, the local fault state of e becomes `permFail`. Because of the definition of the state `permFail`, the operation may require some synchronization with other sites that refer to e. The operation must ensure that no other site can perform operations on the entity. Therefore the operation `Kill` is not guaranteed to succeed.

The example in Snippet 4.2 shows a simple abstraction, yet quite powerful. It basically tries to ensure that all entities in a list eventually fail when one of them fails.

4.3.2 Local failure

Sometimes it is not possible to make an entity fail globally, for instance because a site that is involved in the operation `Kill` has silently crashed. We therefore provide the operation `Break`. The statement `{Break` e`}` has a pure local effect. It makes the entity e fail locally, and forces its fault state to be at least `localFail`.

The first motivation for `Break` is that it is irreversible. Once an entity is permanently failed, even locally, it cannot go back to the fault state `ok`. This is useful when a site triggers a recovery mechanism, based on the state `tempFail`

```
proc {FailAfter E TimeOut}
   proc {Loop L}
      case L of H|T then
         if H==tempFail andthen {WaitTwo {Alarm TimeOut} T}==1
         then {Break E}
         else {Loop T}
         end
      else skip end
   end
in
   thread {Loop {GetFaultStream E}} end
end
```

Snippet 4.3: A failure handler that provokes local failure after a certain duration of temporary failure

of an entity e. Enforcing the failure of e simplifies the task of recovering. The threads blocked because of the failure of e will never wake up, for instance. This is useful if a service is backed up, and at most one instance of the service can run at any given time.

The second motivation is resource management. By making an entity permanently failed, the programmer gives a hook to its memory management system. Threads that block because of the failure will block forever, unless they can be woken up explicitly by other threads. The system can therefore use the permanence of the failure to detect parts of the program (threads and data) that will no longer affect its behavior. Those parts can be safely removed from memory. Some issues about memory management are described in detail in the next section.

Snippet 4.3 shows a small failure handler that can be used together with other failure handlers. Basically it uses a timeout to make an entity locally failed if it remains temporarily failed for a certain time. The timeout duration is specified in the parameter `TimeOut`. Other failure handlers waiting for state `localFail` are thus automatically triggered after the given inactivity duration.

4.4 Failures and memory management

4.4.1 Blocked threads and fault streams

Entity failures have an effect on the memory management of a program. First, a failed entity can make a thread block. If the failure is temporary, that thread must be kept in memory for its possible resumption. As that thread normally refers to the entity, it keeps the entity alive. However, if the failure is permanent, the thread will block forever, unless it is referred to by another living entity. As we already mentioned in Section 4.3.2, a thread blocking

forever can be safely removed from memory.

Something similar happens with fault streams. An entity keeps its own fault stream alive in memory. This guarantees that the threads monitoring the entity do not silently disappear. But the fault stream itself does not keep the entity alive, so the entity can be removed from memory anyway. Once the entity is removed from memory, the fault stream will no longer be kept alive, and the monitoring threads may block forever. In order to clearly reflect that the fault stream has been "disconnected" from the entity, we make the system *close* the stream, i.e., its tail is bound to `nil`. This action is perfectly consistent, since once the entity is gone, the fault stream will no longer be updated.

Finalization. The closing of the fault stream provides a simple and effective *post-mortem finalization* mechanism. The following abstraction executes a procedure `P` once the entity `E` is no longer in memory. The closing of the stream simply lets the loop exit.

```
proc {Finalize E P}
   thread
      for X in {GetFaultStream E} do skip end
      {P}
   end
end
```

This is particularly useful to recollect memory from failed components in a program. A thread monitoring an entity can already remove references to the entity when it fails, and once it is removed from memory, the monitor can perform some extra actions.

4.4.2 Entity resurrection

It is possible for a site to remove an entity e from its memory, even when that entity is still used by other sites. Indeed, if the site owns a proxy for e that is not necessary for the distribution of the entity, it can safely remove the entity's proxy from its memory (see Sections 3.3.4 and 3.3.6). When this happens, the site simply no longer refers to e, which remains alive on a global scale.

Now assume that the entity e was removed from the memory of site a, and that a reference to e is sent again to that site. A new proxy for e is created on a, and that proxy creates a new fault stream for e on a. This reintroduction of e on site a brings a few issues. First, there is no connection between the new fault stream of e and its former fault stream, which was bound to `nil` by the finalization mechanism described above. The instances of the fault stream in memory correspond to different *sessions* of the entity on the site.

Second, it is possible that the fault state of e was `localFail` before e's removal, and `ok` after its reintroduction. This state transition is normally forbidden by the fault model. To avoid this situation, site a should have kept some information about entity e in memory. But keeping that information in memory is unreasonable in general, because site a's memory would grow beyond

any limit. Our proposal is to *not* keep that information, but to implement a specific solution to handle the issue at the application level. An application may use a centralized or distributed repository of valid entities. Any occurrence of a non valid entity can then be discarded. The management of the repository and the choice of which entities to check is thus specific to the application.

4.5 Related work

4.5.1 Java RMI

In Java Remote Method Invocation (RMI), every distributed operation is a method call. The standard way in that language to report a problem inside a method call is the exception mechanism. The fault model thus favors synchronous failure handlers, which are implemented as exception handlers.

4.5.2 Erlang

The power and simplicity of failure handling in Erlang was an inspiration for our work. Erlang provides asynchronous detection of permanent failures between processes [Arm07]. Two processes can be *linked* together. When one of them (say a) terminates normally or because of a failure, the other one (say b) is notified by the runtime system. By default, process b will die if a died because of a failure. However, if b is a *system process*, it will receive a message of the form `{'EXIT',Pid,Why}`, where `Pid` is the identifier of process a, and `Why` is a value that describes the reason why a died. A special built-in turns a process into a system process.

Erlang chose to model all failures as permanent failures, in accordance with its philosophy of "Let it fail". That is, keeping the fault model simple allows the recovery algorithm to be simple as well. This simplicity is very important for correctness. We can see our model as an extension of Erlang's model with temporary failures and with a fault stream. Furthermore, our model is designed for a richer language than Erlang, which only has stationary ports (in our terminology). Chapter 5 will show how to program something similar to process linking in Oz.

4.5.3 The first fault model of Mozart

Our argument against the use of exceptions to handle distribution failures comes from the original fault model used in Oz, which was introduced with the first release of Mozart in 1998. The original model overlaps with the model we propose in this chapter. It was providing much more fault information (most of which was not used in practice) and was supporting both synchronous and asynchronous handlers. The major difference was the ability to define synchronous failure handlers, i.e., handlers that are called when attempting an

operation on a failed entity. The programmer could either ask for an exception or provide a handler procedure that replaces the operation. The failure handler was defined for a given entity and with certain conditions of activation.

Instead of the synchronous handlers, programmers favored a kind asynchronous handler, called a *watcher*. A watcher is a user procedure that is called in a new thread when a failure condition is fulfilled. The fault stream we propose in this paper simply factors out how the system informs the user program. It also avoids race conditions related to the watcher registry system, which could make one miss a fault state transition. And finally, a watcher could not be triggered by a transition to state `ok`. The latter soon revealed to be problematic for handling temporary failures.

An alternative model. The original model is criticized in [GGV04], which proposes an alternative model. That paper proposes something similar to our fault stream and an operation to make an entity fail locally. In order to handle faults, it proposes to explicitly break the transparent distribution of a failed entity. The local representative of the failed entity is disconnected from its peers and is put in a fault state equivalent to `localFail`. Another operation replaces that entity by a fresh new entity. This model has the advantage to avoid blocking threads on failed entities, because you can replace a failed entity by a healthy one. But this replacement introduces inconsistencies in the application's shared memory. We were not able to give a satisfactory semantics that takes into account these inconsistencies.

5

Applications

We present several abstractions that show how to program with the model we proposed in the former chapters. We first show how to hide network delays in a lazy producer/consumer situation, with a bounded buffer. We propose two implementations for the buffer: a fully declarative version, and a version that automatically adapts the buffer size.

We also show how to implement Erlang-like processes in Oz. Process linking and monitoring is very easy to implement. We then provide an abstraction that deals with temporary failures, and guarantees a consensus about failures in a set of processes monitoring each other. The consensus is reached by a vote among the correct processes.

5.1 Distributed lazy producer/consumer

Assume we have a component producing a stream lazily, and sharing that stream with other components, possibly on other sites. From a language point of view, those components simply share a logic variable. Consumers make the variable needed, which awaken the producer. The latter binds the variable to a pair `X|T`, where `T` is computed lazily as well.

This scheme is a nice example of a declarative communication channel between components. Moreover, its performance is not bad: making the variable needed typically costs one message from the consumer to the producer, and binding the variable costs one message in the other direction. So the variable imposes a communication delay of one round-trip per element. Note that this delay is independent from the number of consumers.

```
fun {BoundedBuffer N Xs}
   fun lazy {Deliver Xs Xr}
      case Xs of X|Xt then X|{Deliver Xt thread Xr.2 end} end
   end
in
   {Deliver Xs thread {Drop N Xs} end}
end
```

Snippet 5.1: A first implementation of a bounded buffer

5.1.1 A bounded buffer

In order to avoid the communication delay, one may insert a buffer between the producer and the consumer. The buffer triggers the evaluation of n elements ahead of the consumer. If the number n is well chosen, and the consumer does not run faster than the producer, then the consumer will not wait for reading one element from the stream. The value n is chosen such that the average time for producing one element, together with the communication delay, does not exceed the average time for consuming n elements.

Snippet 5.1 shows an implementation of a bounded buffer which is equivalent to the one proposed in [VH04]. The function `BoundedBuffer` takes as input the size of the buffer n, and the lazy stream `Xs`, and returns a lazy stream `Ys`:

```
Ys={BoundedBuffer N Xs}
```

The value of `Ys` is equal to `Xs`, except that if m elements of `Ys` are computed, $m + n$ elements of `Xs` are computed. The function call `{Drop N Xs}` returns the list `Xs` without its first `N` elements.

Behavior analysis. First, let us notice that calling `BoundedBuffer` on the producer's site will not fit our needs. Indeed, in that case, the lazy computation associated to the output variable `Ys` is on the producer side. When the consumer makes that variable needed, a full round-trip to the producer's site is necessary to trigger the lazy computation and send the value back.

Suppose now that `BoundedBuffer` is called on the consumer's site. When the consumer reads an element, it triggers a local lazy computation which returns immediately, if the element is available. At the same time, the lazy computation triggers the need for an element n positions ahead in the stream. However, when the consumer reads an extra element, the element n positions ahead will be requested *whenever the element before is delivered on the consumer's site*.

To illustrate that behavior, assume we have a producer/consumer pair with a bounded buffer of size $n = 5$. Let us analyze what happens when the consumer reads three elements from the stream. The interactions between both sites are shown in the left picture of Figure 5.1 on the facing page. The arrows to the

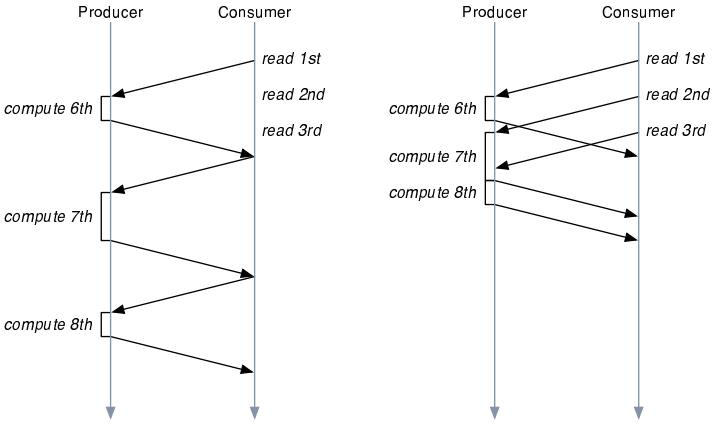


Figure 5.1: Network behavior of two implementations of a bounded buffer of size 5

left represent the messages that make a variable needed, while the arrows to the right are the messages with the binding of the corresponding variable. What we observe is that an element cannot be requested before the list pair containing the former element arrives on the consumer's site.

What we really want is something like the right picture of Figure 5.1. For each element read, the element n positions ahead should be requested as soon as possible. With this behavior, the elements are still produced in a sequential way, but the message round-trips to trigger the production of elements are truly concurrent. In the first behavior, those message round-trips are serialized.

5.1.2 A correct bounded buffer

Snippet 5.2 on the next page shows an implementation that provides the desired behavior. The producer performs the following statement on its site.

```
Es={Encapsulate Xs}
```

The returned value is a pair of variables that will be bound to streams. The first variable is the stream `Xs`, while the second variable is a stream `Rs` that is built by the consumer and read by the producer. For each element appearing on that stream, the producer requests one extra element on the data stream `Xs`. This is done by the thread running procedure `Prepare` on the producer. The consumer makes the following call to get a stream `Ys`.

```
Ys={DecapsulateN N Es}
```

This immediately builds a stream with `N` elements that will be read by the producer. Then, for every element consumed on `Ys`, the stream `Rs` is appended with the statement `Rs=unit|Rt`.

```
fun {Encapsulate Xs}
   proc {Prepare Rs Xs} {Prepare Rs.2 Xs.2} end
   Rs
in
   thread {Prepare Rs Xs} end
   Xs#Rs
end

fun {DecapsulateN N Es}
   fun {Prepend K Xt}
      if K>0 then unit|{Prepend K-1 Xt} else Xt end
   end
   fun lazy {Deliver Xs Rs}
      case Xs of X|Xt then Rt in
         Rs=unit|Rt                  % trigger need at producer
         X|{Deliver Xt Rt}
      end
   end
   Rt
in
   Es.2={Prepend N Rt}               % trigger N elements ahead
   {Deliver Es.1 Rt}
end
```

Snippet 5.2: A correct implementation of a bounded buffer

Let us now check that the behavior of the abstraction corresponds to the picture on the right of Figure 5.1. The arrows from right to left correspond to the bindings `Rs=unit|Rt`, while the arrows from left to right correspond to the bindings of the producer's output stream. Note that the bindings of `Rs` are performed immediately by the consumer. This is because the variables `Rs` are created on the consumer's site, hence the coordinators of those variables are on that site, and variable bindings are performed by the variable's coordinator. A more detailed explanation can be found in Section 8.2.3.

If several consumers are present, the stream can be encapsulated once, and each consumer decapsulates it by applying `DecapsulateN`. The producer will be driven by the consumer that requests the furthest ahead. However, the network behavior involved by the stream `Rs` is more difficult to describe, since the consumers will share that stream. Therefore a binding like `Rs=unit|Rt` might require an intermediate network message to another consumer site, if that other site holds the coordinator of `Rs`.

5.1.3 An adaptive bounded buffer

Snippet 5.3 provides a replacement for the function `DecapsulateN`. The stream is decapsulated on the consumer's site by the statement

```
fun {Decapsulate Es}
   fun lazy {Deliver Xs Rs}
      case Xs of X|Xt then Rt in
         Rs = if {Not {IsDet Xt}} then unit|unit|Rt
              elseif {Not {IsDet Xt.2}} then unit|Rt
              else Rt end
         X|{Deliver Xt Rt}
      end
   end
   Rt
in
   Es.2=unit|Rt
   {Deliver Es.1 Rt}
end
```

Snippet 5.3: An adaptive bounded buffer

```
Ys={Decapsulate Es}
```

This new function no longer takes a buffer size, but instead adapts how elements are requested ahead in order to always have one element ready at the consumer's site.

Let us make a quick comparison between the functions `DecapsulateN` and `Decapsulate`. The main difference is the binding of `Rs`, the second argument of the internal lazy function, which is called `Deliver` in both versions. For each consumed element, the adaptive version checks how many elements are available in front of `Xt`. We use the function `IsDet` which returns **`true`** if its argument is determined, and **`false`** otherwise. If no element is available (`Xt` is not determined), the size of the buffer is increased by triggering the need for two extra elements with the statement `Rs=unit|unit|Rt`. If exactly one element is available (`Xt.2` is not determined yet), we keep the same buffer size by requesting one extra element (`Rs=unit|Rt`). If more than one element is available, we decrease the buffer size by not requesting any extra element (`Rs=Rt`).

This adaptive version of the bounded buffer will work well if the consumer reads the stream at a regular pace.

5.1.4 A batch processing buffer

The reader might be surprised by the solution proposed in the previous sections. The abstractions effectively improve the network behavior of lazy evaluation, but they do it by avoiding the distributed mechanism of lazy evaluation. We were also disappointed by this solution when we realized this. So we came up with a solution that relies on the distributed by-need mechanism.

In order to avoid the sequential "ping-pong" effect illustrated in Figure 5.1, one may let the producer site trigger the evaluation of several elements in a row.

```
proc {BatchBuffer N Xs}
   proc {BatchLoop I Xs}
      if I>0 then {BatchLoop I-1 Xs.2} else
         {WaitNeeded Xs} {BatchLoop N Xs}
      end
   end
in
   thread {BatchLoop 0 Xs} end
end
```

Snippet 5.4: An abstraction that forces the evaluation of a stream in batches

Whenever the first element is requested, an abstraction forces the production of n elements. In other words, we can force the producer to work by *batches*.

The abstraction is shown in Snippet 5.4. One simply has to call

```
{BatchBuffer N Xs}
```

on the producer's site. The procedure creates a thread that detects when an element is needed, and automatically makes the $n-1$ following elements needed. The thread then waits until the element after that batch becomes needed, and requests a new batch.

This abstraction can be used solely, or in combination with the simple bounded buffer given in Snippet 5.1. When used solely, the full round trip delay will occur only once every n elements. If the consumer uses the simple bounded buffer, the round-trip delay can be completely hidden if n is large enough.

5.2 Processes à la Erlang

In the language Erlang, almost everything is a process. A process consists of a port, on which messages can be sent, and a function that processes the messages. A process is created by the primitive `spawn`, and messages are sent with the binary operator `!`:

```
Pid = spawn(F)     % create a process from a function F

Pid ! Msg          % send a message Msg to process Pid
```

The function takes a message from the incoming queue with the statement `receive`. The statement uses pattern matching to specify valid messages, and subsequent actions.

It is pretty easy to write a function `Spawn` in Oz that is similar to the corresponding Erlang primitive. The function, shown in Snippet 5.5 on the next page with an example, creates a port and runs the procedure in its own thread.

```
%% create a process with unary procedure Process
fun {Spawn Process}
   Xs Ys Self={NewPort Xs}
   fun {Loop Xs}
      case Xs of user(M)|Xt then M|{Loop Xt} end
   end
in
   thread Ys={Loop Xs} end
   thread {Process Ys} end
   Self
end

%% send message M to process A
proc {SendProc A M}
   {Send A user(M)}
end
```

Snippet 5.5: Spawning an Erlang-like process in Oz

The procedure takes the stream of messages in argument. The procedure should process the messages in a sequential way. The procedure `SendProc` sends a message `M` to a process `A`. Note that messages are put in a record `user(M)`, in order to distinguish them from system messages that are introduced below.

Here is an example with two processes `A` and `B` sending each other ping-pong messages:

```
proc {ProcessA Xs}
   case Xs of X|Xt then
      case X
      of stop    then skip
      [] ping(P) then {SendProc P pong(A)} {ProcessA Xt}
      end
   end
end
A={Spawn ProcessA}

proc {ProcessB Xs}
   {SendProc A ping(B)}
   case Xs of pong(P)|_ andthen P==A then skip end
end
B={Spawn ProcessB}
```

Process linking. Erlang processes can be *linked* together, such that when one of them terminates abnormally, the other ones die also, unless they are system processes. System processes are explained below. Linking is symmetric, and implements a property which states that a group of processes must crash as soon as one of them crashes. A process `A` adds the process `B` to its link set by evaluating the built-in function `link(B)`.

```
%% link process Self to process A
proc {Link Self A}
   {Send Self link(A)} {Send A link(Self)}
end

%% change the 'system process' flag
proc {SetSystem Self B}
   X in {Send Self system(B X)} {Wait X}
end

%% create a process with unary procedure Process
fun {Spawn Process}
   Xs Ys Self={NewPort Xs} T
   fun {Loop Xs Linkset Sys}
      case Xs of X|Xt then
         case X
         of user(M) then M|{Loop Xt Linkset Sys}
         [] system(B X) then X=unit {Loop Xt Linkset B}
         [] link(A) then {Monitor A} {Loop Xt A|Linkset Sys}
         [] exit(E) then {Notify E Linkset} nil
         [] exit(A E) andthen Sys then X|{Loop Xt Linkset Sys}
         [] exit(A normal) then {Loop Xt Linkset Sys}
         [] exit(A E) then {Kill T} {Notify E Linkset} nil
         end
      end
   end
   proc {Monitor A}
      thread
         if {Member permFail {GetFaultStream A}} then
            {Send Self exit(A crashed)} end
      end
   end
   proc {Notify E Linkset}
      for A in Linkset do {Send A exit(Self E)} end
   end
in
   thread Ys={Loop Xs nil false} end
   thread
      T={Thread.this}
      try {Process Ys} {Send Self exit(normal)}
      catch E then {Send Self exit(E)} end
   end
   Self
end
```

Snippet 5.6: Asymmetric linking and monitoring of processes

In Snippet 5.6 on the facing page we propose a new implementation of `Spawn` that handles linking and system processes. The internal loop of the process handles system messages, and maintains a link set, i.e., a list of processes that are notified of the termination of the current process. The message `link(A)` is sent by the procedure `Link`, and notifies the current process that it is linked to process `A`. The current process adds `A` to its link set, and monitors `A` to detect a failure that `A` itself would not be able to notify.

When the process terminates, it sends the message `exit(E)` to itself, in order to notify its link set. The value `E` describes the reason of the termination. Processes in the link set are notified with the message `exit(Self E)`. The latter message is handled in a different way, depending on whether the current process is a system process. Non-system processes are killed when `E` is not `normal`, while system processes simply receive the message. The message `system(B)` is sent by procedure `SetSystem` by the process itself in order to change its status (system process or not).

5.3 Failure by majority

Failure detectors are extremely useful for writing programs that react to partial failures. Failure handling can be written in a rule-based style. However our detector model is weak in the sense that detectors are not required to be consistent between sites. This can sometimes lead distributed programs to behave strangely, because some sites consider an entity failed, and others don't.

In this section we propose an algorithm that makes a group of N processes find a *consensus* about the failure status of a given process. Group members may themselves fail during the consensus algorithm. However, the algorithm is guaranteed to reach consensus about crashed processes if less than $N/2$ processes crash.

5.3.1 Algorithm

The idea is quite simple. For the sake of simplicity, let us assume that the group wonders about the status of process S. Whenever process P suspects S, it broadcasts $vote(P, +1)$. If it changes its mind about S, it broadcasts $vote(P, -1)$. Every process sums the values received from every other process, and maintains how many of them have a positive account. When this number becomes greater than $N/2$, it means that a majority of processes have suspected S. At that point a message is broadcast to make all correct processes consider S as permanently failed, i.e., `{Break` S`}`. The latter message must be broadcast in a reliable manner, in order to guarantee the consistency between processes.

The algorithm is described in Figure 5.2 on the next page in the style of Guerraoui *et al.* [GR06]. Processes perform actions when some events occur, some of those events being messages. An implementation with objects is given in Snippet 5.7 on page 71. The specificity of the algorithm is that there is

upon S is suspect **do**
broadcast $vote(self, +1)$

upon S is non-suspect **do**
broadcast $vote(self, -1)$

upon event $vote(P, x)$ **do**
$count.P := count.P + x$

upon event $kill(S)$ **do**
execute `{Break` S`}`

upon $\#\{P \mid count.P > 0\} > \frac{N}{2}$ **do**
reliableBroadcast $kill(S)$

Figure 5.2: Majority voting for consensus on failure status of S

only one possible decision, and that decision is taken whenever a majority of processes agreed with that decision. Moreover, the decision is monotonic.

Note. Counting the votes from a given process P consists in summing all the $+1$'s and -1's sent by P. The counter allows to receive votes from P in any order. If all messages from P are received in order, the counter will always belong to the set $\{0, 1\}$.

5.3.2 Correctness

Assume that S has crashed. Because we rely on eventually perfect failure detectors, all correct processes will eventually suspect S forever. So at some point, at least $N/2$ processes will broadcast a positive vote. And all correct processes will eventually sum all the votes broadcast by the positive majority we mentioned. Note that broadcasting the decision is only an optimization in that case.

Now assume that S has not crashed. There may be enough suspicions among the other processes to let one of them observe a majority of positive votes. The latter observation might be temporary if processes change their mind quickly. In that case, broadcasting the decision with a reliable algorithm ensures that all correct processes will eventually consider S as permanently failed.

In order to show the necessity of the final broadcast, let us imagine an extreme case where only one process P observes a majority of positive votes. Such a scenario is depicted in Figure 5.3 on the next page. A group of N_X processes X suspect S, then cancel their suspicion. Their messages reach P faster than another group Y. The group Y has N_Y processes, that also suspect S then revise their judgment. If we have both $N_X, N_Y < N/2$ and $N_X + N_Y > N/2$, then only P will observe a majority of positive votes. If P does not broadcast its observation, its view will not be consistent with the other processes.

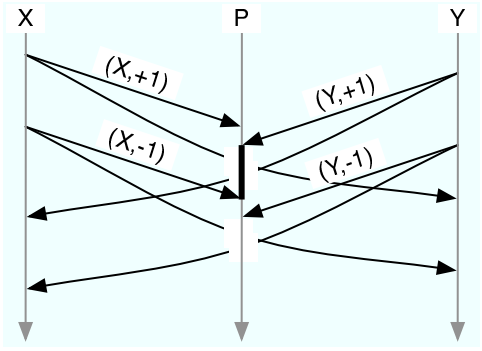


Figure 5.3: A scenario where only one process P observes a majority

5.3.3 The whole code of processes

The process itself is implemented as an object that multiplexes voters. Each process has one voter per other process it monitors. The whole code is given in Snippets 5.8 to 5.12 on pages 72–74.

The class `BaseProcess` is the mother class of all processes. It creates the process' port, and a thread that processes messages. An identifier `id` is also provided. The latter is useful if one wants to use a process as a key in a dictionary.

The class `ProcessWithFailureDetector` extends `BaseProcess` by monitoring other processes. It is initialized with the list of processes, with their identifier and port. A process using this class should implement method `failure()` in order to handle failures. In our example, this method is defined in subclass `MonitoringProcess`.

The classes `BestEffortBroadcast` and `ReliableBroadcast` provide simple methods to broadcast messages to all processes. The latter is *reliable* in the sense that either all correct processes deliver the message, or none of them deliver it. Each process that receives the message broadcast it once, too. This ensures the delivery in case the original sender crashes. This implementation is not efficient, but it fulfills its specification.

The class `MonitoringProcess` is the main class. It multiplexes its voters, and provides all the support they need for communicating. In method `failure`, you can see that the "opinion" of a voter is changed when the failure state of the corresponding process changes.

5.3.4 Variants

The algorithm we gave was kept simple for the sake of explanation. But it is quite flexible, and admits variants, which are easy to implement. Here are a few ideas that can improve the abstraction.

- One may change the number of positive votes that must be reached to trigger the decision. A value of $N/3$ may be considered enough, for instance.
- We have assumed the number of processes to be known and fixed. The algorithm works fine if that number varies over time, and processes regularly update this number. The condition for triggering the decision simply has to be reevaluated.
- One should discard crashed processes when counting votes. One may also discard suspect processes, such that only known correct processes are taken into account. The latter idea requires more attention, because as such, it would allow one process to suspect all other processes.

```
class Voter
   attr
      broadcast       % broadcast procedure
      rbBroadcast     % reliable broadcast procedure
      decide          % decide procedure
      total           % number of processes
      id              % identifier of this process
      opinion         % this process' opinion (true or false)
      votes           % accumulated votes from each voter

   meth init(broadcast:B rbBroadcast:RB decide:D N)
      broadcast := B
      rbBroadcast := RB
      decide := D
      total := N
      id := {NewName}
      opinion := false
      votes := {NewDictionary}
   end

   %% set the process' opinion (true for suspicion)
   meth propose(B)
      if B \= @opinion then
         opinion := B
         {@broadcast vote(@id (if B then 1 else ~1 end))}
      end
   end

   %% receive a vote from Id
   meth vote(Id X)
      @votes.Id := {Dictionary.condGet @votes Id 0} + X
      if X > 0 then N in
         N={Length {Filter {Dictionary.items @votes} IsPos}}
         if N*2 > @total then {@rbBroadcast decide} end
      end
   end

   %% receive decision
   meth decide
      {@decide true}
   end
end

fun {IsPos X} X>0 end
```

Snippet 5.7: Implementation of the majority voting algorithm

```
class BaseProcess
   feat port id

   meth init()
      Xs in
      thread
         try {ForAll Xs self}
         catch _ then {Kill self.port}
         end
      end
      self.port={NewPort Xs}
      self.id={NewName}
   end
end
```

Snippet 5.8: Base class of processes

```
class ProcessWithFailureDetector from BaseProcess
   attr processes

   meth initProcesses(IPs)
      %% IPs is a list of pairs id#port
      processes := {List.toRecord p IPs}
      for Id#P in IPs do
         thread
            {Wait P}
            for X in {GetFaultStream P} do
               {Send self.port failure(Id X)}
            end
         end
      end
   end
end
```

Snippet 5.9: A class for processes that monitor each other

```
class BestEffortBroadcast from ProcessWithFailureDetector
   meth broadcast(M)
      {Record.forAll @processes proc {$ P} {Send P M} end}
   end
end
```

Snippet 5.10: Implementation of best-effort broadcast

```
class ReliableBroadcast from BestEffortBroadcast
   attr delivered

   meth initProcesses(IPs)
      BestEffortBroadcast,initProcesses(IPs)
      delivered := nil
      {self CheckDelivered}
   end

   meth CheckDelivered
      %% some kind of 'garbage collection' on delivered
      delivered := unit|{List.takeWhile @delivered
                         fun {$ Id} Id \= unit end}
      thread
         {Delay 360000} {Send self.port CheckDelivered}
      end
   end

   meth rbBroadcast(M)
      %% note: unicity of messages is guaranteed by user
      {self broadcast(rbDeliver(M))}
   end

   meth rbDeliver(M)
      if {Not {Member M @delivered}} then
         delivered := M|@delivered
         {Send self.port M}
         {self broadcast(rbDeliver(M))}
      end
   end
end
```

Snippet 5.11: Implementation of an “eager” reliable broadcast

```
class MonitoringProcess from ReliableBroadcast
   attr voters

   meth initProcesses(IPs)
      N={Length Ps}
   in
      ReliableBroadcast,init(IPs)
      voters := {Record.mapInd {List.toRecord v IPs}
                 fun {$ I P}
                    proc {B M}
                       {self broadcast(voting(I M))}
                    end
                    proc {RB M}
                       {self rbBroadcast(voting(I M))}
                    end
                    proc {D B}
                       if B then {self kill(I)} end
                    end
                 in
                    {New Voter init(broadcast:B
                                    rbBroadcast:RB
                                    decide:D
                                    N)}
                 end}
   end

   %% relay a message for a voter
   meth voting(I M)
      {@voters.I M}
   end

   %% failure detector notification, maybe change opinion
   meth failure(I State)
      {@voters.I propose(State \= ok)}
   end

   %% decide whether a process has failed
   meth kill(I)
      {Break @processes.I} {Kill @processes.I}
      voters := {AdjoinAt @voters I proc {$ _} skip end}
   end
end
```

Snippet 5.12: Main class, with one voter per process in the group

6

Language semantics

We now give a formal support to the language concepts we presented in Chapters 2, 3 and 4. This chapter defines an operational semantics to Oz without taking distribution into account. The next chapter presents a refinement of that semantics, which models distribution, network, and failures. The refinement also give a semantics to the annotation system and the failure handling primitives.

Section 6.1 defines how to translate a program in Full Oz into an equivalent program in Kernel Oz. Section 6.2 gives the notations and basic ingredients of the semantic definitions. Sections 6.3 details the semantics of the declarative part of the kernel language, while Section 6.4 gives the semantics of the non-declarative part of the language.

6.1 Full language to kernel language

Every Oz program is equivalent to a program in Kernel Oz. In Chapter 2, we have introduced the kernel language, and syntactic sugar of common idioms in the full language. We now see how to formally translate an Oz program into an equivalent Kernel Oz program. The kernel language is given by the grammar in Figure 6.1. Both the declarative and non-declarative parts of the language are given in the grammar.

The translation is defined by the relation $\Rightarrow$, which reduces statements to simpler statements. The kernel program equivalent to a given Oz program is defined as the fixpoint of the program by the relation. This relation is structural: one can reduce a statement inside another statement.

For the rest of the section, D denotes a declaration (statement or identifier), E an expression, P a pattern, S a statement, SE a statement or expression, and X and Y identifiers.

$$
\begin{aligned}
S ::= {} & \texttt{skip} \mid S_1 S_2 \mid \texttt{thread}\ S\ \texttt{end} \\
& \mid \texttt{local}\ X\ \texttt{in}\ S\ \texttt{end} \\
& \mid X\texttt{=}Y \mid X\texttt{=}f(Y_1 \ldots Y_n) \\
& \mid \texttt{if}\ X\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end} \\
& \mid \texttt{case}\ X\ \texttt{of}\ f(Y_1 \ldots Y_n)\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end} \\
& \mid \{\texttt{WaitNeeded}\ X\} \\
& \mid \texttt{proc}\ \{X\ Y_1 \ldots Y_n\}\ S\ \texttt{end} \mid \{X\ Y_1 \ldots Y_n\} \\
& \mid \texttt{try}\ S_1\ \texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end} \mid \texttt{raise}\ X\ \texttt{end} \mid \{\texttt{FailedValue}\ X\ Y\} \\
& \mid X\texttt{=!!}Y \\
& \mid \{\texttt{NewCell}\ X\ Y\} \mid X_0\texttt{=}Y\texttt{:=}X_1
\end{aligned}
$$

Figure 6.1: Grammar of Kernel Oz

Expanding declarations. The following rules split up declarations into declared identifiers and initializing statements. It simplifies declarations as "**local** `X=foo` **in**". We assume that the statements that appear in the declaration D have already been reduced to kernel statements.

$$
\begin{aligned}
D\ \texttt{in}\ SE &\Rightarrow \texttt{local}\ D\ \texttt{in}\ SE\ \texttt{end} \\
\texttt{local}\ D\ \texttt{in}\ SE\ \texttt{end} &\Rightarrow \texttt{local}\ \mathsf{decl}(D)\ \texttt{in}\ \mathsf{stmt}(D)\ SE\ \texttt{end} \\
\texttt{local}\ X_1\ X_2 \ldots X_n\ \texttt{in}\ SE\ \texttt{end} &\Rightarrow \texttt{local}\ X_1\ \texttt{in} \\
&\qquad \texttt{local}\ X_2 \ldots X_n\ \texttt{in}\ SE\ \texttt{end} \\
&\quad \texttt{end}
\end{aligned}
$$

The functions decl and stmt respectively return the declared identifiers and the statements of a declaration. The function ident returns the set of identifiers in a pattern; each identifier is declared at most once. Those functions are defined as

$$
\begin{aligned}
\mathsf{decl}(X) &= \{X\} \\
\mathsf{decl}(P\texttt{=}E) &= \mathsf{ident}(P) \\
\mathsf{decl}(P\texttt{=}E_1\texttt{:=}E_2) &= \mathsf{ident}(P) \\
\mathsf{decl}(\texttt{proc}\ \{X\ Y_1 \ldots Y_n\}\ S\ \texttt{end}) &= \{X\} \\
\mathsf{decl}(S) &= \emptyset \\
\mathsf{decl}(D_1 \ldots D_n) &= \mathsf{decl}(D_1) \cup \ldots \cup \mathsf{decl}(D_n)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{ident}(X) &= \{X\} \\
\mathsf{ident}(f(P_1 \ldots P_n)) &= \mathsf{ident}(P_1) \cup \cdots \cup \mathsf{ident}(P_n)
\end{aligned}
$$

$$\begin{aligned}\mathsf{stmt}(X) &= \epsilon\\ \mathsf{stmt}(S) &= S\\ \mathsf{stmt}(D_1 \dots D_n) &= \mathsf{stmt}(D_1) \dots \mathsf{stmt}(D_n)\end{aligned}$$

Expanding nested expressions. Those are the kernel statements that contain an expression E in place of an identifier. The reduction introduces an identifier X, and expands the evaluation of E before evaluating the statement itself. The identifier X is chosen such as to not occur in the original statement. Notice that in the procedure call, the first non-identifier is expanded. We assume that $m, n \geq 0$.

$$\begin{aligned}E{=}E' &\Rightarrow \texttt{local}\ X{=}E\ \texttt{in}\ X{=}E'\ \texttt{end}\\ \texttt{if}\ E\ \texttt{then}\dots\texttt{end} &\Rightarrow \texttt{local}\ X{=}E\ \texttt{in}\ (\texttt{if}\ X\ \texttt{then}\dots\texttt{end})\ \texttt{end}\\ \texttt{case}\ E\ \texttt{of}\dots\texttt{end} &\Rightarrow \texttt{local}\ X{=}E\ \texttt{in}\ (\texttt{case}\ X\ \texttt{of}\dots\texttt{end})\ \texttt{end}\\ \texttt{proc}\ \{E\dots\}\ S\ \texttt{end} &\Rightarrow \texttt{local}\ X{=}E\ \texttt{in}\ (\texttt{proc}\ \{X\dots\}\ S\ \texttt{end})\ \texttt{end}\\ \{Y_1 \dots Y_m\ E\ E_1 \dots E_n\} &\Rightarrow \texttt{local}\ X{=}E\ \texttt{in}\ \{Y_1 \dots Y_m\ X\ E_1 \dots E_n\}\ \texttt{end}\\ \texttt{raise}\ E\ \texttt{end} &\Rightarrow \texttt{local}\ X{=}E\ \texttt{in}\ (\texttt{raise}\ X\ \texttt{end})\ \texttt{end}\end{aligned}$$

Expanding expressions. Function definitions are expanded to procedures. The extra parameter X is chosen to not occur in a free position in E.

$$\texttt{fun}\ \{\dots\}\ E\ \texttt{end} \Rightarrow \texttt{proc}\ \{\dots\ X\}\ X{=}E\ \texttt{end}$$

Then we expand all the statements of the form $X{=}E$. The expansion often brings the assignment to X inside the language constructs, which sometimes declares new identifiers Y_i. If X occurs in those declarations, then we substitute this occurrence of X by another identifier. The result of this substitution is denoted Y_i^* or E^*.

$$\begin{aligned}X{=}(S\ E) &\Rightarrow S\ X{=}E\\ X{=}\texttt{thread}\ E\ \texttt{end} &\Rightarrow \texttt{thread}\ X{=}E\ \texttt{end}\\ X{=}\texttt{local}\ Y\ \texttt{in}\ E\ \texttt{end} &\Rightarrow \texttt{local}\ Y^*\ \texttt{in}\ X{=}E^*\ \texttt{end}\\ X{=}E_1{=}E_2 &\Rightarrow X{=}E_1\ X{=}E_2\\ X{=}\texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2\ \texttt{end} &\Rightarrow \texttt{if}\ E\ \texttt{then}\ X{=}E_1\ \texttt{else}\ X{=}E_2\ \texttt{end}\\ X{=}\texttt{case}\ E\ \texttt{of}\ f(Y_1 \dots Y_n)\ \texttt{then}\ E_1\ \texttt{else}\ E_2\ \texttt{end} &\Rightarrow \texttt{case}\ E\ \texttt{of}\ f(Y_1^* \dots Y_n^*)\ \texttt{then}\ X{=}E_1^*\ \texttt{else}\ X{=}E_2\ \texttt{end}\\ X{=}\texttt{proc}\ \{\$\dots\}\ S\ \texttt{end} &\Rightarrow \texttt{proc}\ \{X\dots\}\ S\ \texttt{end}\\ X{=}\{E\ E_1 \dots E_n\} &\Rightarrow \{E\ E_1 \dots E_n\ X\}\\ X{=}\texttt{try}\ E_1\ \texttt{catch}\ Y\ \texttt{then}\ E_2\ \texttt{end} &\Rightarrow \texttt{try}\ X{=}E_1\ \texttt{catch}\ Y^*\ \texttt{then}\ X{=}E_2^*\ \texttt{end}\\ X{=}\texttt{raise}\ E\ \texttt{end} &\Rightarrow \texttt{raise}\ E\ \texttt{end}\end{aligned}$$

The `lazy` expansion. Lazy functions can be defined by using **`fun lazy`** instead of **`fun`** in their definition. The simplest way to expand this construct is to create a thread that synchronizes on the demand, then evaluates the function's body expression. We assume that the parameter X does not occur in a free position in E.

$$\begin{array}{l}\textbf{fun}\ \texttt{lazy}\ \{\ldots\}\ E\ \textbf{end} \Rightarrow \textbf{proc}\ \{\ldots\ X\} \\ \qquad\qquad\qquad\qquad\quad \textbf{thread}\ \{\texttt{WaitNeeded}\ X\}\ X{=}E\ \textbf{end} \\ \qquad\qquad\qquad\qquad \textbf{end}\end{array}$$

While being correct, this expansion may suffer a slight performance overhead, especially if the function is recursive. The overhead comes from the fact that every recursive call creates a new thread. Recursive calls in tail position do not need this extra thread. For those, one may let the current thread suspend. This is correct as long as the initial call to the function is in a different thread. The following expansion optimizes tail recursive calls.

$$\begin{array}{l}\textbf{fun}\ \texttt{lazy}\ \{F\ X_1 \ldots X_n\}\ E\ \textbf{end} \Rightarrow \textbf{local}\ F'\ \textbf{in} \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{proc}\ \{F'\ X_1 \ldots X_n\ X\} \\ \qquad\qquad\qquad\qquad\qquad\qquad \{\texttt{WaitNeeded}\ X\}\ (X{=}E)^* \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{end} \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{fun}\ \{F\ X_1 \ldots X_n\} \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{thread}\ \{F'\ X_1 \ldots X_n\}\ \textbf{end} \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{end} \\ \qquad\qquad\qquad\qquad\qquad \textbf{end}\end{array}$$

The identifier F' is chosen to not occur in the definition of F. The extra operation $(X{=}E)^*$ expands the statement $X{=}E$, and replaces every tail call to F by a similar call to F'.

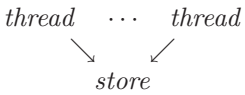
The `$` expansion. The main use of the `$` sign is in expressions that define a procedure, or in a procedure call. At some point such an expression E will be reduced in a statement of the form $X{=}E$. The following rules show how to reduce such a statement. $P(X)$ denotes a pattern containing an identifier X, and $P(\texttt{\$})$ is the same pattern with X replaced by `$`.

$$X{=}\textbf{proc}\ \{\texttt{\$}\ldots\}\ S\ \textbf{end} \Rightarrow \textbf{proc}\ \{X\ldots\}\ S\ \textbf{end}$$
$$X{=}\textbf{fun}\ \{\texttt{\$}\ldots\}\ E\ \textbf{end} \Rightarrow \textbf{fun}\ \{X\ldots\}\ E\ \textbf{end}$$
$$X{=}\{E_1 \ldots E_m\ P(\texttt{\$})\ E_{m+1} \ldots E_n\} \Rightarrow \{E_1 \ldots E_m\ P(X)\ E_{m+1} \ldots E_n\}$$

More linguistic abstractions. The full language provides even more statements, like class definitions, functor definitions, support for constraints, etc. We do not show how to expand those in this work. Material can be found in the book [VH04], and the documentation of Mozart [Moz99].

6.2 Basics of the semantics

Here we give the basics of the operational semantics of the language. The semantic rules prescribe how a running program can go from one state to another. A program state is called a *configuration*. A configuration consists of a set of threads connected to a shared store:



The thread is the basic unit of sequential computation. A computation consists of a sequence of computation steps, each of which transforms a configuration into another configuration. At each step, a thread is chosen, and executes an atomic operation. The choice of the thread is nondeterministic among all the executable threads in that configuration. Thread execution follow the interleaving semantics.

6.2.1 The store

The store is a single-assignment store (or constraint store), extended with first-class procedures, mutable entities, and a few other specific extensions [Smo95, VH04]. The extensions will be introduced step by step, together with their corresponding reductions rules. Those extensions are grouped together under the term *predicate store*.

The constraint store contains variable assignments made by the program. Assignments are between variables ($x{=}y$), or between variables and values ($x{=}v$). The constraint store is a conjunction of such assignments. It has the property of being monotonic, in the sense that one can only *add* assignments; existing assignments cannot be removed.

Store entailment. The constraint store has a logic nature, it can entail information that is not directly present in the store. For instance, the store $x{=}3 \wedge x{=}y$ entails $y{=}3$. We denote a store by σ, and a basic relation like an equality by β. The statement $\sigma \models \beta$ means that the store σ entails β. We assume that the store conjunction is associative, commutative, and has neutral element $\top$, which also denotes the empty store.

What the constraint store entails is defined by the following inference rules. The rules are given with premises on top of a horizontal line, and a conclusion below. The horizontal line is not shown when the premises are true. The very first rule states that the store entails at least what it contains, and in particular, that adding information in the store never reduces entailment.

$$\sigma \wedge \beta \models \beta \tag{6.1}$$

The next rules are specific to the equality relation. Rules (6.2) simply reflect that equality is reflexive, symmetric, and transitive. The metavariables t, u, v

can be either variables or values. The values we consider here are either simple values, or records.

$$\sigma \models t{=}t \qquad \frac{\sigma \models u{=}v}{\sigma \models v{=}u} \qquad \frac{\sigma \models t{=}u \quad \sigma \models u{=}v}{\sigma \models t{=}v} \tag{6.2}$$

We now define rules for record equality. Two records are equal if and only if they have identical labels, arities, and their fields are pairwise equal. The following two rules establish the "positive" side of this statement.

$$\frac{\sigma \models u_1{=}v_1 \quad \cdots \quad \sigma \models u_n{=}v_n}{\sigma \models f(u_1 \ldots u_n){=}f(v_1 \ldots v_n)} \tag{6.3}$$

$$\frac{\sigma \models f(u_1 \ldots u_n){=}f(v_1 \ldots v_n)}{\sigma \models u_i{=}v_i} \quad 1 \leq i \leq n \tag{6.4}$$

The constraint store can also *disentail* some equalities, i.e., inferring that they are false. The following rules state explicitly that records with different labels, arities, or different corresponding fields are unequal.

$$\frac{f{\neq}g \quad \text{or} \quad m{\neq}n}{\sigma \models f(u_1 \ldots u_m){\neq}g(v_1 \ldots v_n)} \tag{6.5}$$

$$\frac{\sigma \models u_i{\neq}v_i}{\sigma \models f(u_1 \ldots u_n){\neq}f(v_1 \ldots v_n)} \quad 1 \leq i \leq n \tag{6.6}$$

Determinacy. We can generalize a bit store entailment, in order to introduce derived concepts like determinacy. We say a variable x is determined by a store σ if σ entails that it is equal to a given value. We note this as $\sigma \models \mathit{det}(x)$. If the store cannot infer the value of the variable, we say that the variable is free.

$$\frac{\sigma \models x{=}v}{\sigma \models \mathit{det}(x)} \quad \text{for a certain value } v \tag{6.7}$$

Ask and tell. The two basic operations on a store are called *ask* and *tell*. The ask operation queries the store to know whether a given constraint is entailed or disentailed. Asking β on store σ returns a positive answer if $\sigma \models \beta$, a negative answer if $\sigma \models \neg\beta$. There is no answer otherwise. The monotonicity of the store guarantees that the answer of an ask never changes.

The tell operation adds a basic constraint to a store, provided that the store remains consistent. The store becomes inconsistent as soon as it infers something like 1=2, for instance. Telling β to σ updates the store to $\sigma \wedge \beta$. The rules that update the store are written such that they never make the store inconsistent. If an inconsistency could be introduced by a program statement, that statement should fail.

Predicate store. The predicate store is subject to the principle of substitution by equals. The following inference rule states that an instance of predicate p is entailed by the store if the store contains a similar predicate whose arguments are pairwise equal.

$$\frac{\sigma \models u_1{=}v_1 \quad \cdots \quad \sigma \models u_n{=}v_n}{\sigma \wedge p(u_1,\ldots,u_n) \models p(v_1,\ldots,v_n)} \tag{6.8}$$

Contrary to the constraint store, elements of the predicate store can be removed, or replaced. The valid ways to update the predicate store depends on each predicate, and is defined by the semantic rules.

6.2.2 Structural rules

The semantics are given by transition rules (or *reduction* rules[1]) that describe valid computation steps. The rules have the form

$$\frac{\mathcal{T} \;\|\; \sigma}{\mathcal{T}' \;\|\; \sigma'} \quad \text{if } C$$

It states that a configuration with a multiset of threads $\mathcal{T}$ and store σ can be reduced to the configuration with threads $\mathcal{T}'$ and store σ', provided the condition C is fulfilled. We often write the left-hand side of the rule as a pattern, so that a configuration must match the pattern for the rule to be applicable. The disjoint union of multisets is written with commas, and singletons are written without curly braces. For instance, "$T_1, \mathcal{T}, T_2$" stands for $\{T_1\} \uplus \mathcal{T} \uplus \{T_2\}$. There is no ambiguity because of the thread syntax.

The following two rules are convenient for simplifying the expression of the rules. The first one expresses the relative isolation of concurrent threads: a subset of the thread may reduce without directly affecting the other threads.

$$\frac{\mathcal{T},\mathcal{U} \;\|\; \sigma}{\mathcal{T}',\mathcal{U} \;\|\; \sigma'} \quad \text{if} \quad \frac{\mathcal{T} \;\|\; \sigma}{\mathcal{T}' \;\|\; \sigma'} \tag{6.9}$$

The second rule states that stores can be considered up to equivalence. This allows to choose the most convenient representation for a store in a reduction rule.

$$\frac{\mathcal{T} \;\|\; \sigma}{\mathcal{T} \;\|\; \sigma'} \quad \text{if } \sigma \text{ and } \sigma' \text{ are equivalent} \tag{6.10}$$

Store equivalence is defined as follows. Let us first consider the constraint store. Two stores $\sigma = \beta_1 \wedge \cdots \wedge \beta_n$ and $\sigma' = \beta'_1 \wedge \cdots \wedge \beta'_{n'}$ are equivalent if

$$\sigma \models \beta'_i \quad \text{for every } i, \text{ and} \qquad \sigma' \models \beta_j \quad \text{for every } j. \tag{6.11}$$

[1] This expression comes from the chemical analogy of transition rules, where the execution takes a statement, and *reduces* it to a simpler statement.

The other part of the store follows a similar rule, except that each instance of a predicate in σ must correspond to *exactly* one predicate in σ'. This is necessary for some predicates, like the one that defines the current state of a cell, which must occur exactly once per cell in the store.

6.3 Declarative subset of the language

6.3.1 Sequential and concurrent execution

A thread is a sequence of statements $S_1 S_2 \dots S_n$. Parentheses are introduced to avoid ambiguities when necessary. The empty thread is written (). The abstract syntax of threads can thus be defined as

$$T ::= () \mid S\,T \tag{6.12}$$

The empty thread reduces to an empty multiset of threads. A nonempty thread reduces by reducing its first statement. The latter rule will again simplify the expression of rules.

$$\frac{() \;\|\; }{\sigma \;\|\; \sigma} \qquad\qquad \frac{S\,T \;\|\; S'\,T}{\sigma \;\|\; \sigma'} \text{ if } \frac{S \;\|\; S'}{\sigma \;\|\; \sigma'} \tag{6.13}$$

The empty statement, sequential composition, and thread statement are tied to the notion of thread. For those rules we have to show explicitly how they modify the structure of the threads. Notice that the latter creates a new thread with the statement S only.

$$\frac{\texttt{skip}\;T \;\|\; T}{\sigma \;\|\; \sigma} \qquad \frac{(S_1\,S_2)\,T \;\|\; S_1\,(S_2\,T)}{\sigma \;\|\; \sigma} \qquad \frac{\texttt{thread}\;S\;\texttt{end}\;T \;\|\; T, S}{\sigma \;\|\; \sigma} \tag{6.14}$$

6.3.2 Variable introduction

The **`local`** statement creates a new variable in the store, and make the declared identifier correspond to that variable. Instead of maintaining an explicit mapping between identifiers and variables, we directly substitute the declared identifier by its corresponding variable. The notation $S[X/x]$ stands for the substitution of X by x in S. The substitution takes care of lexical scope issues.

$$\frac{\texttt{local}\;X\;\texttt{in}\;S\;\texttt{end} \;\|\; S[X/x]}{\sigma \;\|\; \sigma} \quad \text{where } x \text{ is a fresh variable} \tag{6.15}$$

The condition of the rule requires x to be a fresh variable. A fresh variable is a variable that does not appear anywhere in the initial configuration. This can written formally, but we have chosen to keep the rule more readable.

Variable substitution. The identifier substitution operation is quite usual. Assume that θ denotes the substitution $[X/x]$. Let χ denote an identifier or a variable.

$$\chi\theta = \begin{cases} x & \text{if } \chi = X \\ \chi & \text{otherwise} \end{cases} \tag{6.16}$$

We now define the substitution inductively on the syntax of statements. The following statements do not involve lexical scoping.

$$(\texttt{skip})\theta = \texttt{skip} \tag{6.17}$$
$$(S_1\ S_2)\theta = S_1\theta\ S_2\theta \tag{6.18}$$
$$(\texttt{thread}\ S\ \texttt{end})\theta = \texttt{thread}\ S\theta\ \texttt{end} \tag{6.19}$$
$$(\chi_1\texttt{=}\chi_2)\theta = \chi_1\theta\texttt{=}\chi_2\theta \tag{6.20}$$
$$(\chi\texttt{=}c)\theta = \chi\theta\texttt{=}c \tag{6.21}$$
$$(\chi\texttt{=}f(\chi_1 \ldots \chi_n))\theta = \chi\theta\texttt{=}f(\chi_1\theta \ldots \chi_n\theta) \tag{6.22}$$
$$(\texttt{if}\ \chi\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end})\theta = \texttt{if}\ \chi\theta\ \texttt{then}\ S_1\theta\ \texttt{else}\ S_2\theta\ \texttt{end} \tag{6.23}$$
$$(\{\chi\ \chi_1 \ldots \chi_n\})\theta = \{\chi\theta\ \chi_1\theta \ldots \chi_n\theta\} \tag{6.24}$$
$$(\texttt{raise}\ \chi\ \texttt{end})\theta = \texttt{raise}\ \chi\theta\ \texttt{end} \tag{6.25}$$

In the following equations, we assume that the lexical scope introduced by the statement does not catch X, i.e., X is different from the identifiers $Y, Y_1, \ldots, Y_n$.

$$(\texttt{local}\ Y\ \texttt{in}\ S\ \texttt{end})\theta = \texttt{local}\ Y\ \texttt{in}\ S\theta\ \texttt{end} \tag{6.26}$$
$$\begin{pmatrix}\texttt{case}\ \chi\ \texttt{of}\ f(Y_1 \ldots Y_n) \\ \quad\texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}\end{pmatrix}\theta = \begin{array}{l}\texttt{case}\ \chi\theta\ \texttt{of}\ f(Y_1 \ldots Y_n) \\ \quad\texttt{then}\ S_1\theta\ \texttt{else}\ S_2\theta\ \texttt{end}\end{array} \tag{6.27}$$
$$(\texttt{proc}\ \{\chi\ Y_1 \ldots Y_n\}\ S\ \texttt{end})\theta = \texttt{proc}\ \{\chi\theta\ Y_1 \ldots Y_n\}\ S\theta\ \texttt{end} \tag{6.28}$$
$$(\texttt{try}\ S_1\ \texttt{catch}\ Y\ \texttt{then}\ S_2\ \texttt{end})\theta = \texttt{try}\ S_1\theta\ \texttt{catch}\ Y\ \texttt{then}\ S_2\theta\ \texttt{end} \tag{6.29}$$

We now define the substitution when X is caught by the lexical scope of the statements. We assume that $X \in \{X_1, \ldots, X_n\}$.

$$(\texttt{local}\ X\ \texttt{in}\ S\ \texttt{end})\theta = \texttt{local}\ X\ \texttt{in}\ S\ \texttt{end} \tag{6.30}$$
$$\begin{pmatrix}\texttt{case}\ \chi\ \texttt{of}\ f(X_1 \ldots X_n) \\ \quad\texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}\end{pmatrix}\theta = \begin{array}{l}\texttt{case}\ \chi\theta\ \texttt{of}\ f(X_1 \ldots X_n) \\ \quad\texttt{then}\ S_1\ \texttt{else}\ S_2\theta\ \texttt{end}\end{array} \tag{6.31}$$
$$(\texttt{proc}\ \{\chi\ X_1 \ldots X_n\}\ S\ \texttt{end})\theta = \texttt{proc}\ \{\chi\theta\ X_1 \ldots X_n\}\ S\ \texttt{end} \tag{6.32}$$
$$(\texttt{try}\ S_1\ \texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end})\theta = \texttt{try}\ S_1\theta\ \texttt{catch}\ X\ \texttt{then}\ S_2\ \texttt{end} \tag{6.33}$$

6.3.3 Unification

The unification operation in Oz imposes equality between two terms. It incrementally tells basic constraints to the store until the equality is entailed or disentailed by the store. The operational semantics of unification is therefore

non atomic. This lack of atomicity permits a realistic extension of unification in the distributed case.

The following two rules terminate the unification when it is either entailed, or disentailed. The statement **`fail`** is used for the sake of readability; it is shorthand for **`raise failure end`**, which raises a failure exception.

$$\frac{u{=}v \;\|\; \texttt{skip}}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models u{=}v \tag{6.34}$$

$$\frac{u{=}v \;\|\; \texttt{fail}}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models u{\neq}v \tag{6.35}$$

We then give the rule that incrementally tells basic constraints to the store. Those basic constraints are necessary for the unification to succeed. They are of the form $x{=}t$, where x is not determined by the store yet, and t is either a variable or a value.

$$\frac{u{=}v \;\|\; u{=}v}{\sigma \;\|\; \sigma \wedge x{=}t} \quad \text{if } \sigma \wedge u{=}v \models x{=}t \text{ and } \sigma \not\models det(x) \tag{6.36}$$

There exists an optional simplification rule, that rewrites a unification as another one. This simplification does not change the effect of unification, but it allows an implementation to simplify it.

$$\frac{u{=}v \;\|\; u'{=}v'}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \wedge u{=}v \models u'{=}v' \text{ and } \sigma \wedge u'{=}v' \models u{=}v \tag{6.37}$$

Example. Executing $x{=}f(y)$ with the store $\sigma \equiv x{=}f(x_1) \wedge x_1 = 2$ tells $y{=}2$ to the store, then reduce to **`skip`**. Indeed, the first reduction applies since the store inference rules give

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\sigma \wedge x{=}f(y) \models x{=}f(y)}{\sigma \wedge x{=}f(y) \models f(y){=}x} \qquad \sigma \wedge x{=}f(y) \models x{=}f(x_1)}{\sigma \wedge x{=}f(y) \models f(y){=}f(x_1)}}{\sigma \wedge x{=}f(y) \models y{=}x_1} \qquad \sigma \wedge x{=}f(y) \models x_1{=}2}{\sigma \wedge x{=}f(y) \models y{=}2}$$

The rule leads to the store $\sigma' \equiv \sigma \wedge y{=}2$, which entails $x{=}f(y)$:

$$\dfrac{\sigma' \models x{=}f(x_1) \qquad \dfrac{\dfrac{\sigma' \models x_1{=}2 \qquad \dfrac{\sigma' \models y{=}2}{\sigma' \models 2{=}y}}{\sigma' \models x_1{=}y}}{\sigma' \models f(x_1){=}f(y)}}{\sigma' \models x{=}f(y)}$$

6.3.4 Conditional statements

Those statements perform an *ask* operation on the store, and possibly block until a condition is entailed or disentailed.

The `if` statement. The classical conditional statement reduces depending on the value of its condition variable. The statement waits until the variable equals **`true`** or **`false`**, then reduces accordingly:

$$\frac{\texttt{if}\ x\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end} \;\|\; S_1}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models x{=}\texttt{true} \tag{6.38}$$

$$\frac{\texttt{if}\ x\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end} \;\|\; S_2}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models x{=}\texttt{false} \tag{6.39}$$

The value of x is usually determined by a boolean function, like a comparison operator. If x is different from **`true`** and **`false`**, the statement reduces by raising an exception (see Section).

The `case` statement. It can be seen as a linguistic abstraction for pattern matching, expressed in terms of a conditional statement, variable introduction, and record operations `Label` and `Arity`. However the concept is important enough to be presented with its semantic rules.

$$\frac{\begin{array}{l}\texttt{case}\ x\ \texttt{of}\ f(X_1 \ldots X_n)\\ \quad\texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}\end{array} \;\|\; S_1[X_1/x_1]\cdots[X_n/x_n]}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models x{=}f(x_1 \ldots x_n) \tag{6.40}$$

$$\frac{\begin{array}{l}\texttt{case}\ x\ \texttt{of}\ f(X_1 \ldots X_n)\\ \quad\texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}\end{array} \;\|\; S_2}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models x{\neq}f(x_1 \ldots x_n) \tag{6.41}$$

The pattern matches if the store entails the equality $x{=}f(x_1 \ldots x_n)$, for some variables $x_1, \ldots, x_n$. In case of a match, the statement reduces to S_1, where the identifiers X_i are substituted by the corresponding variables x_i in x. If the store disentails any such equality, the statement reduces to S_2. If the store does not contain enough information to decide one way or another, then the statement cannot reduce.

Waiting for determinacy. We have defined above what it means for a store to determine a variable. Waiting for the determination of a variable is the most direct way to show the dataflow behavior of variables. It can be expressed explicitly with the unary procedure `Wait`. Its semantics is extremely simple: it reduces to **`skip`** once its argument is determined.

$$\frac{\{\texttt{Wait}\ x\} \;\|\; \texttt{skip}}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models \mathit{det}(x) \tag{6.42}$$

6.3.5 Names and procedures

Names are unforgeable constants, and have therefore no textual representation. They are useful to give a unique identity to a language entity like a procedure or a cell. But they can also be used as first-class values by a programmer. Such a value can be confined by lexical scope to the implementation of a data structure, for keeping a feature hidden to the user. For instance, names are used to define private methods in a class, which are by default only accessible from within the class.

Names are created explicitly by the operation `NewName`. Its semantics are given by the following reduction rule. Every fresh name is guaranteed to be different from all other existing names and values. Names are created in a way similar to variables. The semantic statement $x{=}\xi$ is obtained by semantic rule reduction only. This reduction clearly separates the name creation from the binding of the variable x.

$$\frac{\texttt{\{NewName } x\texttt{\}} \;\|\; x{=}\xi}{\sigma \;\|\; \sigma} \quad \text{where } \xi \text{ is a fresh name} \tag{6.43}$$

Procedures. The **proc** statement creates a procedure in the store. The procedure value consists in a name ξ that is associated to a statement abstraction in the procedure store by the pair $\xi : \lambda X_1 \ldots X_n.S$. All the free identifiers of S are in the set $\{X_1, \ldots, X_n\}$. The name gives the procedure its identity.

Procedure application performs an ask to the store. For applying procedure p, p must be equal to a name ξ that is associated to a statement abstraction. Procedure application thus blocks if p is not determined by the store. Once the procedure is known, the call reduces to the abstracted statement, where each parameter is substituted by the corresponding argument in the call.

$$\frac{\textbf{proc } \{p\ X_1 \ldots X_n\}\ S\ \textbf{end} \;\|\; p{=}\xi}{\sigma \;\|\; \sigma \wedge \xi{:}\lambda X_1 \ldots X_n.S} \quad \xi \text{ a fresh name} \tag{6.44}$$

$$\frac{\{p\ x_1 \ldots x_n\} \;\|\; S[X_1/x_1] \cdots [X_n/x_n]}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models p{=}\xi \wedge \xi{:}\lambda X_1 \ldots X_n.S \tag{6.45}$$

6.3.6 By-need synchronization

Lazy evaluation, or demand-driven computation, is possible in Oz via the by-need synchronization mechanism. It works as follows. In a producer-consumer scheme, the producer and the consumer are in separate threads, and share a logic variable x. The producer simply blocks until a consumer requires x to be determined in order to reduce. The mechanism that allows the producer to detect the need of a consumer is called *by-need synchronization*. Note that the mechanism is very general, and allows several producers and consumers for a single variable.

The semantics is defined in terms of *ask* and *tell* on the *by-need store*. The latter is an extension of the constraint store, and is monotonic as well, which makes this language concept fully declarative. The predicate $needed(x)$ is used to synchronize producers and consumers: it is automatically told to the store by the consumer if the determinacy of x is required for its reduction. The producer uses the unary procedure `WaitNeeded` to synchronize on the entailment of the predicate by the store.

In our proposal, we consider that determined variables are needed by convention. This simplifies the behavior of a variable. We identify three states, which are ordered in this way: free, needed, and determined. State transitions follow that order, which ensures the monotonicity of the store. Moreover, this convention disambiguates a producer-consumer situation where a consumer would bind the shared variable. The variable automatically becomes needed, and the producer is woken up. We provide this property with the following inference rule.

$$\frac{\sigma \models det(x)}{\sigma \models needed(x)} \tag{6.46}$$

Now consider a statement S. We define $needed(S)$ as the set of variables which must be determined for S to be executable.

$$x \in needed(S) \quad \text{iff} \quad \begin{cases} \text{for every store } \sigma\text{:} \\ \text{if } S \text{ is executable with } \sigma\text{, then } \sigma \models det(x) \end{cases} \tag{6.47}$$

The condition can also be expressed as: the statement S cannot reduce in a configuration where x is not determined. The definition directly applies to the `Wait` statement: $x \in needed(\texttt{\{Wait } x\texttt{\}})$. The variable x is also needed by the statements "**if** $x \ldots$" and "**case** $x \ldots$".

The first reduction rule below describes how the predicate $needed(x)$ is told to the store. The second rule states that the statement `{WaitNeeded` x`}` asks the store for the predicate $needed(x)$, and reduces to **skip** once it is entailed.

$$\frac{S \;\|\; \sigma}{S \;\|\; \sigma \wedge needed(x)} \quad \text{if } x \in needed(S), \text{ and } \sigma \not\models needed(x) \tag{6.48}$$

$$\frac{\texttt{\{WaitNeeded } x\texttt{\}} \;\|\; \sigma}{\textbf{skip} \;\|\; \sigma} \quad \text{if } \sigma \models needed(x) \tag{6.49}$$

6.4 Nondeclarative extensions

6.4.1 Nondeterministic wait

The function `WaitTwo` takes two arguments, and returns the number of the argument that is determined (`1` or `2`). The returned value is nondeterministic in case both arguments are determined. It can be used for merging streams,

for instance.

$$\frac{\texttt{\{WaitTwo}\ x\ y\ z\texttt{\}} \;\|\; z\texttt{=1}}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models det(x) \tag{6.50}$$

$$\frac{\texttt{\{WaitTwo}\ x\ y\ z\texttt{\}} \;\|\; z\texttt{=2}}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models det(y) \tag{6.51}$$

6.4.2 Exception handling

We first introduce the **try** statement. In order to simplify the management of the scope defined by the statement, we consider a **catch** statement, which can only be obtained by the reduction of the first rule below. The **catch** statement itself reduces to **skip**. These two rules model all executions where no exception is raised.

$$\frac{\textbf{try}\ S_1\ \textbf{catch}\ X\ \textbf{then}\ S_2\ \textbf{end} \;\|\; S_1\ (\textbf{catch}\ X\ \textbf{then}\ S_2\ \textbf{end})}{\sigma \;\|\; \sigma} \tag{6.52}$$

$$\frac{\textbf{catch}\ X\ \textbf{then}\ S_2\ \textbf{end} \;\|\; \textbf{skip}}{\sigma \;\|\; \sigma} \tag{6.53}$$

Consider now the **raise** statement. This statement is either written explicitly in the program, or is obtained by a reduction rule in case of an error. For instance, an **if** statement reduces to a **raise** statement if the condition variable is not of type boolean. The effect of the **raise** statement is to skip all statements after it, except a **catch** statement.

$$\frac{\textbf{raise}\ x\ \textbf{end}\ (\textbf{catch}\ X\ \textbf{then}\ S_2\ \textbf{end})\ T \;\|\; S_2[X/x]\ T}{\sigma \;\|\; \sigma} \tag{6.54}$$

$$\frac{\textbf{raise}\ x\ \textbf{end}\ S\ T \;\|\; \textbf{raise}\ x\ \textbf{end}\ T}{\sigma \;\|\; \sigma} \quad \text{if } S \text{ is not a } \textbf{catch} \text{ statement} \tag{6.55}$$

This simple model works fine with any number of nested **try** statements, and reflects well that the scope defined by the statement only covers the current threads. An exception in a thread cannot be caught by another thread.

Failed values. Those special values provide a way to transmit exceptions from one thread to another. A failed value y encapsulates an exception x, and is represented in the store by y=*failed*(x). It is created by the operation `FailedValue`.

$$\frac{\texttt{\{FailedValue}\ x\ y\texttt{\}} \;\|\; y\text{=}\mathit{failed}(x)}{\sigma \;\|\; \sigma} \tag{6.56}$$

If a statement S needs a failed value, S immediately reduces by raising the exception. The exception is also raised if the statement tries to bind the value.

$$\frac{S \;\|\; \texttt{raise}\ x\ \texttt{end}}{\sigma \;\|\; \sigma} \quad \text{if } y \in \mathit{needed}(S) \text{ and } \sigma \models y{=}\mathit{failed}(x) \tag{6.57}$$

$$\frac{u{=}v \;\|\; \texttt{raise}\ x\ \texttt{end}}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \wedge u{=}v \models \mathit{det}(y) \text{ and } \sigma \models y{=}\mathit{failed}(x) \tag{6.58}$$

6.4.3 Read-only views

Oz provides a useful concept for protecting data structures from accidental bindings from the user. This protection allows a user to read a variable without being able to bind it. The idea is to pair two variables x and y by making y a read-only view of x. We write this pairing as $y{=}\mathit{view}(x)$. Such a pair is created by the "bang bang" operator `!!`.

$$\frac{y{=}\texttt{!!}x \;\|\; y{=}z}{\sigma \;\|\; \sigma \wedge z{=}\mathit{view}(x)} \quad \text{where } z \text{ is a fresh variable} \tag{6.59}$$

Once the variable x is determined, being a view of x implies being equal to x. This property is given by the following inference rule. Note that it could be used to drop views from the store, and replace them by equalities: when x is determined, $y{=}\mathit{view}(x)$ is replaced by $y{=}x$.

$$\frac{\sigma \models y{=}\mathit{view}(x) \qquad \sigma \models \mathit{det}(x)}{\sigma \models y{=}x} \tag{6.60}$$

Preventing unification. As read-only views cannot be determined before their variable, we have to strengthen the condition for binding a variable during unification, and make sure that we never bind a read-only view of a variable. The rule (6.36) is rewritten as

$$\frac{u{=}v \;\|\; u{=}v}{\sigma \;\|\; \sigma \wedge x{=}t} \quad \text{if} \begin{cases} \sigma \wedge u{=}v \models x{=}t \\ \sigma \not\models \mathit{det}(x), \quad \text{and for all } y,\ \sigma \not\models x{=}\mathit{view}(y) \end{cases} \tag{6.61}$$

Views and by-need synchronization. Read-only views can be used to protect lazy computations, provided that a variable becomes needed when its view is needed. The following inference rule on the store does the job.

$$\frac{\sigma \models y{=}\mathit{view}(x) \qquad \sigma \models \mathit{needed}(y)}{\sigma \models \mathit{needed}(x)} \tag{6.62}$$

Just like a determined variable is needed, one can expect that any attempt to determine a view by unification makes the view needed. Although it does not

exactly fit our definition of needing a variable, we propose the following rule, which makes views needed in case of unification.

$$\frac{u{=}v \;\|\; u{=}v}{\sigma \;\|\; \sigma \wedge needed(y)} \quad \text{if} \begin{cases} \sigma \wedge u{=}v \models det(y) \\ \sigma \models y{=}view(x), \quad \text{and } \sigma \not\models needed(y) \end{cases} \tag{6.63}$$

6.4.4 State

All stateful entities can be built on top of cells. The semantics of cells will therefore serve as a reference for all stateful entities with synchronous operations: arrays, dictionaries, etc. Ports can also be built on top of cells. However we will consider a fully asynchronous version of the `Send` operation, which will be given a specific distributed semantics.

A cell is semantically defined as a name associated to a state in the stateful store. If ξ is the name of the cell, the stateful store contains the predicate $\xi{:}x$, where the variable x is the current state of the cell. The creation of a cell consists in creating a name, and adding an initial state for it in the stateful store.

$$\frac{\texttt{\{NewCell}\ x\ c\texttt{\}} \;\|\; c{=}\xi}{\sigma \;\|\; \sigma \wedge \xi{:}x} \quad \text{where } \xi \text{ is a fresh name} \tag{6.64}$$

Just like names, the statement reduces to unifying the cell variable to the name: $c{=}\xi$. This separates the creation of the cell from the binding of the variable c.

Synchronous operations. All synchronous operations can be modeled as a cell *exchange* operation. This operation possibly changes the state of the entity, and returns its former state. The semantics of the operation is given by the following reduction rule.

$$\frac{x{=}c\texttt{:=}y \;\|\; x{=}w}{\sigma \wedge \xi{:}w \;\|\; \sigma \wedge \xi{:}y} \quad \text{if } \sigma \models c{=}\xi \tag{6.65}$$

Asynchronous operations. The operation `Send` on port will serve as a reference for asynchronous operations. Its specificity is that the statement reduction and the state update are not necessarily made together. The statement reduction corresponds to the message begin sent, and the state update to the message being received.

Let us first propose a definition of ports on top of cells. The port is defined as a cell that contains a part of the stream of received messages. This reference to is used to extend the stream as new messages arrive. The stream of messages does not reveal the tail itself, but a read-only view instead. This guarantees that only the port abstraction can add messages to the stream.

```
proc {NewPort S P}
   T in P={NewCell T} S=!!T
end

proc {Send P X}
   T in X|!!T = P := T
end
```

The semantics of `NewPort` is derived from its code. However the semantics given by this definition of `Send` is not satisfactory. This definition is actually synchronous. It works perfectly in a centralized setting. It has the observable property that all messages sent from a given thread are received in the order they were sent. In other words, each thread imposes a partial order on the reception of its own messages. We call this property the *sender ordering*.

Let us propose a fully asynchronous definition of `Send`. The definition below allow messages to arrive in any order. We will use this definition as a reference.

```
proc {Send P X}
   thread T in X|!!T = P := T end
end
```

Let us provide semantic rules that reflect well the semantics of the asynchronous `Send`. The thread created is modeled as a special thread $n \Leftarrow x$, which represents the message being sent. This special thread reduces upon message reception, which adds the message to the message stream.

$$\frac{\texttt{\{Send}\ p\ x\texttt{\}}\ T \;\|\; T, \xi \Leftarrow x}{\sigma \qquad\quad \sigma} \quad \text{if } \sigma \models p{=}\xi \tag{6.66}$$

$$\frac{\xi \Leftarrow x \;\|\;}{\sigma \wedge \xi{:}t \;\|\; \sigma \wedge \xi{:}t' \wedge t{=}x | s \wedge s{=}\mathit{view}(t')} \quad \text{if } s,\ t' \text{ are fresh variables} \tag{6.67}$$

Note that the real semantics of `Send` satisfies the sender ordering property. This property is clearly not satisfied by our semantic rules. It would require to model one message queue per port and per thread. We have chosen to keep the semantics simple, as we believe that this improved model is not significant for the operation itself.

7

Distributed semantics

The operational semantics we give in this chapter *refines* the centralized semantics given in the former chapter. The refinement is defined in the following way: every distributed configuration $\mathcal{D}$ maps to a centralized configuration $\mathcal{C}$, and every distributed reduction $\mathcal{D} \xrightarrow{d} \mathcal{D}'$ maps to a valid centralized reduction $\mathcal{C} \xrightarrow{c} \mathcal{C}'$. The identity reduction, where $\mathcal{C}{=}\mathcal{C}'$, is considered valid. This kind of property is usually visualized by a commutative diagram like

$$\begin{array}{ll} \mathcal{C} \xrightarrow{c} \mathcal{C}' & \text{(centralized)} \\ \uparrow \quad \uparrow & \\ \mathcal{D} \xrightarrow{d} \mathcal{D}' & \text{(distributed)} \end{array}$$

The semantics should reflect some aspects of the distribution, like partial failures and network latency. Those are necessary to reason about the program. The semantics should also reflect the distribution strategy of entities. Not all entities are distributed the same way. For instance, stateful entities allow at least three schemes (centralized, mobile, and replicated). Each strategy should be clearly identifiable in the semantics, but they all must map to the same centralized semantics.

That being said, the semantics should abstract as much as possible the details which are not relevant for the programmer. For instance, we do not describe how failures are detected in practice, but we give conditions that failure detectors must satisfy. Neither do we specify how communication takes place over the network, how data are serialized, or how Oz names are guaranteed unique across machines. Those issues are supposed to be solved for the programmer. What the semantics give are the elements that the programmer must be aware of, like network and site failures, and the elements that are under the programmer's control, like the distribution strategy and failure modes of language entities.

Sections 7.1 and 7.2 define store extensions that reflect the sites, the network, and how entities are distributed among sites. The semantics of `Annotate` is also given there. Sections 7.3 and 7.4 give the distributed semantics for the declarative and nondeclarative parts of the kernel language, respectively. Section 7.5 gives the semantics of the fault stream and the operations `Kill` and `Break`. Section 7.6 gives the mapping from distributed to centralized semantics.

7.1 Reflecting network and site behavior

The basic principle of a distributed semantics is to incorporate some information about the network and sites in the system. The semantics implicitly provides a formal model of the program environment. The advantage is to reflect site and network behavior at the programming language level. So that the programmer can explain or predict the effect of environment changes on his or her program.

7.1.1 Locality

The very first consequence of distributing a program is to introduce a notion of locality. Each site in the system has only a partial view of the whole store. Though the network transparency aims at abstracting this fact, it is essential for reflecting performance and failure issues. We consider that a distributed configuration is like a centralized configuration, where each thread and each store element is tagged with a site identifier. Consider for instance

$$\frac{(y{=}x{+}1\ z{=}y{*}2)_a, (w{=}z{>}100)_b}{(x{=}42)_a \wedge (x{=}42)_c \wedge \ldots}$$

The first thread runs on site a, while the second thread runs on site b. The store contains at least the constraint $x{=}42$, which is present on both sites a and c. Dropping the site index gives an equivalent centralized store.

The local configuration of a site a in the system is simply the restriction of the distributed configuration to the elements indexed by a, that we denote $\mathcal{T}|_a/\sigma|_a$. The operator $|_a$ ("at a") is defined by the following equations. Both β and γ denote predicates, but γ has either a subscript different from a or no subscript (like the predicate $a{\leftrightarrow}b$ defined in the next section).

$$\begin{array}{rlrl} (\mathcal{T},\mathcal{T}')|_a &= \mathcal{T}|_a, \mathcal{T}'|_a & (\sigma\wedge\sigma')|_a &= \sigma|_a \wedge \sigma'|_a \\ T_a|_a &= T & \beta_a|_a &= \beta \\ T_b|_a &= \emptyset & \gamma|_a &= \top \end{array} \qquad (7.1)$$

7.1.2 Network failures

Let us first enrich the store with information about network links. A network link between sites a and b is operational when the predicate $a{\leftrightarrow}b$ is present in

the store. We consider for the sake of simplicity that those links are bidirectional.

$$\frac{\sigma \models a{\leftrightarrow}b}{\sigma \models b{\leftrightarrow}a} \tag{7.2}$$

The rules below temporarily cut network links, and restores them. We consider that those rules are triggered by the system itself. They define valid state transitions for the model of the environment.

$$\frac{}{\sigma \wedge a{\leftrightarrow}b \parallel \sigma} \qquad \frac{}{\sigma \parallel \sigma \wedge a{\leftrightarrow}b} \quad \text{if } \sigma \not\models a{\leftrightarrow}b \tag{7.3}$$

7.1.3 Site failures

The notion of locality above is expressed in terms of site. We now model site failures in the semantics. Remember that a site failure is of the kind *crash-stop*. Its effect is simply to drop the part of the configuration that is specific to that site. It is described by the global reduction rule

$$\frac{\mathcal{T} \parallel \mathcal{T} \downarrow a}{\sigma \parallel \sigma \downarrow a} \tag{7.4}$$

where the operator $\downarrow a$ ("down a") is defined by the following equations, with the same convention as above for γ.

$$\begin{array}{rr} (\mathcal{T}, \mathcal{T}') \downarrow a = (\mathcal{T} \downarrow a), (\mathcal{T}' \downarrow a) & (\sigma \wedge \sigma') \downarrow a = (\sigma' \downarrow a) \wedge (\sigma' \downarrow a) \\ T_a \downarrow a = \emptyset & \beta_a \downarrow a = \top \\ T_b \downarrow a = T_b & \gamma \downarrow a = \gamma \end{array} \tag{7.5}$$

Site failures have no synchronous effect on other sites. Therefore the operators $\downarrow$ and $|$ have the following properties. The first states that a site failure removes everything that is specific to the site, and the second states that other sites are not affected by the failure.

$$(\mathcal{C} \downarrow a)|_a = \top \tag{7.6}$$

$$(\mathcal{C} \downarrow a)|_b = \mathcal{C}|_b \tag{7.7}$$

7.2 Reflecting entity behavior

In order to handle distributed entities, we introduce three extra ingredients that reflect parts of their behavior. Those elements are mostly independent from the type of entity.

7.2.1 Entity failures

In order to reflect entity failures, we introduce the predicate $\mathsf{alive}(e)$ in the predicate store. This predicate is put in the store at the creation of e, and its absence means the permanent failure of e. It may occur at most once per entity in the whole store, and is localized on a given site (its coordination site). Note that it applies to variables as entities, and that $\mathsf{alive}(x)$ is not equivalent to $\mathsf{alive}(\xi)$, even if $x{=}\xi$ in the store. The principle of substitution by equals does not apply to the predicate alive.

Remember that each site maintains a current fault state for each entity in the system. We assume that on every site a, the store entails one equality like $(\mathsf{fstate}(e){=}s)_a$, which states that a considers entity e to be in fault state s. The precise definition of how the store entails that fact and modifies it, is given in Section 7.5. The principle of substitution by equals does not apply to fstate.

An entity e is *correct* if and only if the distributed store contains $\mathsf{alive}(e)_a$ for some site a. Removing the predicate automatically makes e permanently failed. For a site b to perform an operation on e, we will require e to be correct, accessible, and not locally failed on b. Note that this condition is necessary but not sufficient. In order to abstract a bit this condition, we introduce the predicate $\mathsf{correct}$ on b, which is defined by the following equation.

$$\mathsf{correct}(e,a)_b \equiv \mathsf{alive}(e)_a \wedge a{\leftrightarrow}b \wedge (\mathsf{fstate}(e){=}\mathsf{ok})_b \tag{7.8}$$

Let us comment a bit on each of the conditions required by $\mathsf{correct}(e,a)_b$.

- The failure of site a causes the predicate $\mathsf{alive}(e)_a$ to be dropped from the store; this effectively prevents any further operation on e that would require it to be correct. This property is enough to model the blocking behavior of operations on failed entities.

- The second condition states that site b must be able to communicate with site a. This is because making a consistent update of the entity generally requires some synchronization with the coordination site of the entity.

- The third condition states that b considers e to be in fault state ok. This means that b must be consistent with itself. This is necessary, as b may consider e to be locally failed.

7.2.2 Entity annotations

Some entities have several alternatives for their distributed semantics. Which alternative is used depends on an entity's *annotation*. Every entity annotation is visible in the store as a predicate, like $\mathsf{stationary}(e)$. Note that the predicate is not localized to a site, which means that any site referring to the entity should know its annotations.

Let us consider protocol annotations. We define the predicates stationary, migratory, replicated, variable, reply, immediate, eager, lazy. They are idempotent and mutually inconsistent for a given entity. For instance, we have

$$\begin{aligned} \mathsf{variable}(x) \wedge \mathsf{variable}(x) &\equiv \mathsf{variable}(x) \\ \mathsf{stationary}(\xi) \wedge \mathsf{replicated}(\xi) &\equiv \bot \end{aligned}$$

Access architecture annotations are defined in a similar way. Reference consistency protocol annotations as well, except that an annotation specifies a subset of the provided protocols. Note also that annotations may be inconsistent if they are applied to the wrong type of entity. A cell cannot be annotated with variable, for instance.

We also use the predicate $\mathsf{annot}(e, v)$ as an alternative notation to state that entity e is annotated with v. Each predicate mentioned above corresponds to exactly one value v:

$$\begin{aligned} \mathsf{annot}(e, \texttt{stationary}) &\equiv \mathsf{stationary}(e) \\ \mathsf{annot}(e, \texttt{migratory}) &\equiv \mathsf{migratory}(e) \\ &\vdots \\ \mathsf{annot}(e, \mathsf{lazy}) &\equiv \mathsf{lazy}(e) \end{aligned}$$

Setting annotations. Here we define how annotations are set on entities. The first rule tells the annotation of the entity to the store. There is a similar rule, which we don't mention here, that raises an exception if the annotation is inconsistent.

$$\frac{(\{\texttt{Annotate}\ e\ t\})_a \;\|\; (\textbf{skip})_a}{\sigma \;\|\; \sigma \wedge \mathsf{annot}(e, v)} \quad \text{if} \begin{cases} \sigma \models \mathsf{alive}(e)_a \\ \sigma|_a \models t{=}v \text{ for a value } v \\ \sigma \wedge \mathsf{annot}(e, v) \text{ is consistent} \end{cases} \tag{7.9}$$

The second rule defines the effect of a *default* annotation: if the entity is shared by more than one site and no annotation was specified, a default one is picked at the home site of the entity.

$$\frac{}{\sigma \;\|\; \sigma \wedge \mathsf{annot}(e, v)} \quad \text{if} \begin{cases} \sigma|_b \text{ refers to } e \\ \sigma \models \mathsf{alive}(e)_a \\ v \text{ is default for } e \text{ on } a \\ \sigma \wedge \mathsf{annot}(e, v) \text{ is consistent} \end{cases} \tag{7.10}$$

The condition "$\sigma|_b$ refers to e" means that e occurs in a predicate or an equality in the store $\sigma|_b$.

7.3 Declarative kernel language

We now give the distributed semantics of language statements.

7.3.1 Purely local reductions

The rules that do not modify the store require very small adaptation to the distributed case. Basically the threads must be localized on a site, and the condition must be evaluated with the local store $\sigma|_a$, where a is the site of the reduced thread.

This is the case for the sequential and concurrent composition. Notice that the **`thread`** statement creates a thread on the site where the statement reduces. The conditional statements and procedure application are also extended in this way.

7.3.2 Variable introduction and binding

The rules that introduce and bind variables require an extra adaptation. The creation of a variable x on a site a automatically introduces the predicate $\mathsf{alive}(x)_a$ in the store:

$$\frac{(\texttt{local}\ X\ \texttt{in}\ S\ \texttt{end})_a \;\|\; (S[X/x])_a}{\sigma \;\|\; \sigma \wedge \mathsf{alive}(x)_a} \quad x \text{ fresh variable} \tag{7.11}$$

This predicate is necessary for binding the variable, as shown in the rules below. The first rule binds x on its coordination site first, while the second rule is responsible for propagating the basic constraint from the coordination site to the other sites. Note that the binding $x{=}t$ proposed by site b depends on its local store $\sigma|_b$.

$$\frac{(u{=}v)_b \;\|\; (u{=}v)_b}{\sigma \;\|\; \sigma \wedge (x{=}t)_a} \quad \text{if} \begin{cases} \sigma|_b \wedge u{=}v \models x{=}t \\ \sigma|_a \not\models det(x) \\ \sigma \models \mathsf{correct}(x,a)_b \end{cases} \tag{7.12}$$

$$\frac{\quad \|\quad}{\sigma \wedge (x{=}t)_a \;\|\; \sigma \wedge (x{=}t)_a \wedge (x{=}t)_b} \quad \text{if} \begin{cases} \sigma|_b \text{ refers to } x \\ \sigma|_b \not\models x{=}t \\ \sigma \models \mathsf{correct}(x,a)_b \end{cases} \tag{7.13}$$

Notice how the latter rule propagates references to t on all sites b that refer to the variable x.

7.3.3 Procedure creation and copying

Procedures do not require much adaptation, since they are values. However, we should model the copying of the value from site to site. The value is copied at

most once per site, which keeps the local stores consistent. In the rules below, P is the abstraction $\lambda X_1 \ldots X_n.S$.

$$\frac{(\texttt{proc}\ \{p\ X_1 \ldots X_n\}\ S\ \texttt{end})_a \quad\|\quad (p{=}\xi)_a}{\sigma \quad\|\quad \sigma \wedge (\xi{:}P)_a} \quad \xi \text{ fresh name} \tag{7.14}$$

$$\frac{}{\sigma \wedge (\xi{:}P)_a \quad\|\quad \sigma \wedge (\xi{:}P)_a \wedge (\xi{:}P)_b} \quad \text{if} \begin{cases} \sigma|_b \text{ refers to } \xi \\ \sigma \models \mathsf{eager}(\xi) \wedge a{\leftrightarrow}b \\ \sigma \not\models (\xi{:}P)_b \end{cases} \tag{7.15}$$

The latter rule propagates the abstraction P on all sites that refer to ξ. As a consequence, all the references in the procedure body S are also propagated on those sites.

Lazy copying. If the procedure is annotated with `lazy`, the copy of the abstraction should be done lazily. The reduction rule for copying is similar to the one above, except that it should be reducible only when it is needed on site b, i.e., when a thread on b tries to call it.

$$\frac{S_b \quad\|\quad S_b}{\sigma \wedge (\xi{:}P)_a \quad\|\quad \sigma \wedge (\xi{:}P)_a \wedge (\xi{:}P)_b} \quad \text{if} \begin{cases} p \in \mathit{needed}(S), \quad \sigma|_b \models p{=}\xi \\ \sigma \models \mathsf{lazy}(\xi) \wedge a{\leftrightarrow}b \\ \sigma \not\models (\xi{:}P)_b \end{cases} \tag{7.16}$$

7.3.4 By-need synchronization

This mechanism is easy to extend in the distributed case, since making variable x needed consists in telling the constraint $\mathit{needed}(x)$ everywhere in the system. The two existing rules are extended as purely local rules, such that $\mathit{needed}(x)$ is told and asked locally. The additional rules below propagate the predicate $\mathit{needed}(x)$ to all sites via the coordination site, provided the variable is correct.

$$\frac{}{\sigma \quad\|\quad \sigma \wedge \mathit{needed}(x)_a} \quad \text{if} \begin{cases} \sigma \models \mathsf{correct}(x,a)_b \wedge \mathit{needed}(x)_b \\ \sigma \not\models \mathit{needed}(x)_a \end{cases} \tag{7.17}$$

$$\frac{}{\sigma \quad\|\quad \sigma \wedge \mathit{needed}(x)_b} \quad \text{if} \begin{cases} \sigma \models \mathsf{correct}(x,a)_b \wedge \mathit{needed}(x)_a \\ \sigma \not\models \mathit{needed}(x)_b \end{cases} \tag{7.18}$$

7.4 Nondeclarative extensions

We now complete the distributed semantics of Oz by extending the semantics of nondeclarative language features to the distributed case. Like in the centralized case, those features admit a semantics that is mostly compositional with respect to the declarative part of the language.

7.4.1 Exception handling and read-only views

The exception mechanism interacts with thread execution. Its reduction rules are extended to purely local reductions. Failed values are also reduced locally, and are copied from site to site just like ordinary values.

Read-only views are also handled locally. The basic constraint $x{=}\mathit{view}(y)$ is handled like an ordinary binding, and copied on all sites that refer to x. The binding rule (7.12) is extended with the extra condition

$$\text{for all } y,\ \sigma|_a \not\models x{=}\mathit{view}(y).$$

7.4.2 State

Stateful entities are somewhat richer when it comes to their distribution. As we have seen already, several strategies are possible for maintaining their state. We consider three strategies here, and each has its own semantics: stationary state, migratory state, and replicated state. The properties of those strategies have been discussed in Chapter 3. Our concern in this chapter is that all variants are refinements of the centralized semantics.

Let us first extend the cell creation semantics. The new reduction rule is pretty straightforward: we locate the state on the creation site, together with the predicate $\mathsf{alive}(\xi)$. The cell is distributed once two sites at least refer to the name ξ.

$$\frac{(\{\texttt{NewCell}\ x\ c\})_a \quad\Big\|\quad (c{=}\xi)_a}{\sigma \quad\Big\|\quad \sigma \wedge \mathsf{alive}(\xi)_a \wedge (\xi{:}x)_a} \quad n \text{ fresh name} \tag{7.19}$$

Before we go into the details of the state operations, we have to describe how the state "becomes" distributed among sites. In the case of stationary or migratory state, nothing special is needed. The state is already present on the right site. Replicated state needs some extra support for distributing the state. All we need is one rule that copies the state from its home site a to every other site b that refers to the entity.

$$\frac{}{\sigma \wedge (\xi{:}x)_a \quad\Big\|\quad \sigma \wedge (\xi{:}x)_a \wedge (\xi{:}x)_b} \quad \text{if} \begin{cases} \sigma|_b \text{ refers to } \xi \\ \sigma \models \mathsf{correct}(\xi, a)_b \wedge \mathsf{replicated}(\xi) \\ \sigma \not\models (\xi{:}x)_b \end{cases} \tag{7.20}$$

Synchronous operations. We now give three semantic rules for the cell exchange operation, each rule reflecting the cell's possible distribution strategy. The first rule shows a stationary cell: the state is located at site a. The operation is performed at a.

The second rule shows a migratory cell. The operation reduces once the state move to b, coming from another site b'. The semantics does not tell how b and b' are chosen, this is left to the actual protocol. But there is an interesting

case when $b = b'$: the state remains on b and can be updated without any network operation. This is the "caching" behavior of the migratory state.

The third rule shows a cell whose state is replicated on several sites. In the rule, the symbol $*$ denotes the set of sites that have a copy of the cell's state. One can see that updating the state is costly: it requires the home site of the cell to communicate with all the state replicas. However, if an operation does not change the state, it can be performed on the local copy of the state.

$$\frac{(x{=}c:{=}y)_b \;\|\; (x{=}w)_b}{\sigma \wedge (\xi{:}w)_a \;\|\; \sigma \wedge (\xi{:}y)_a} \quad \text{if} \begin{cases} \sigma|_b \models c{=}\xi \\ \sigma \models \mathsf{correct}(\xi, a)_b \wedge \mathsf{stationary}(\xi) \end{cases} \tag{7.21}$$

$$\frac{(x{=}c:{=}y)_b \;\|\; (x{=}w)_b}{\sigma \wedge (\xi{:}w)_{b'} \;\|\; \sigma \wedge (\xi{:}y)_b} \quad \text{if} \begin{cases} \sigma|_b \models c{=}\xi \\ \sigma \models \mathsf{correct}(\xi, a)_b \wedge \mathsf{migratory}(\xi) \\ \sigma \models b'{\leftrightarrow}b \end{cases} \tag{7.22}$$

$$\frac{(x{=}c:{=}y)_b \;\|\; (x{=}w)_b}{\sigma \wedge (\xi{:}w)_* \;\|\; \sigma \wedge (\xi{:}y)_*} \quad \text{if} \begin{cases} \sigma|_b \models c{=}\xi \\ \sigma \models \mathsf{correct}(\xi, a)_b \wedge \mathsf{replicated}(\xi) \\ \sigma \models a{\leftrightarrow}* \end{cases} \tag{7.23}$$

All those rules have an interesting special case. If site b has the state and performs a read operation, none of the other sites is affected, and the state can be read locally. The only condition is that the local fault state of the entity must be ok. For this case all rules can be simplified to

$$\frac{(x{=}@c)_b \;\|\; (x{=}w)_b}{\sigma \wedge (\xi{:}w)_b \;\|\; \sigma \wedge (\xi{:}w)_b} \quad \text{if} \begin{cases} \sigma|_b \models c{=}\xi \\ \sigma \models (\mathsf{fstate}(\xi){=}\mathsf{ok})_b \end{cases} \tag{7.24}$$

Asynchronous operations. Just like in the centralized semantics, the Send operation reduces immediately by sending a message $p{\Leftarrow}x$.

$$\frac{(\{\mathsf{Send}\ p\ x\}\ T)_b \;\|\; T_b, (\xi{\Leftarrow}x)_b}{\sigma \;\|\; \sigma} \quad \text{if } \sigma|_b \models p{=}\xi \tag{7.25}$$

The first rule below shows the case of the stationary port. The message is received by the coordination site of the entity. Once delivered, the operation is performed, and the binding of the stream is visible globally. The second rule considers a mobile port. In that case, the message is kept locally until the state comes at the site; the operation is then performed locally. The latter rule considers a replicated port. All the copies of the state are atomically changed.

$$\frac{(\xi{\Leftarrow}x)_b \;\|\;}{\sigma \wedge (\xi{:}t)_a \;\|\; \begin{array}{l}\sigma \wedge (t{=}x|t')_a \wedge (\xi{:}t')_a \wedge \\ \mathsf{alive}(t')_a \wedge (t'{=}\mathit{view}())_a\end{array}} \quad \text{if} \begin{cases} t' \text{ is a fresh variable} \\ \sigma \models \mathsf{correct}(\xi, a)_b \\ \sigma \models \mathsf{stationary}(\xi) \end{cases} \tag{7.26}$$

$$\frac{(\xi{\Leftarrow}x)_{b'} \;\|}{\sigma \wedge (\xi{:}t)_b \;\|\; \begin{array}{l}\sigma \wedge (t{=}x|t')_b \wedge (\xi{:}t')_{b'} \wedge \\ \mathsf{alive}(t')_{b'} \wedge (t'{=}\mathit{view}())_{b'}\end{array}} \quad \text{if} \begin{cases} t' \text{ is a fresh variable} \\ \sigma \models \mathsf{correct}(\xi, a)_b \\ \sigma \models \mathsf{migratory}(\xi) \\ \sigma \models \mathsf{correct}(t, b)_{b'} \end{cases} \tag{7.27}$$

$$\frac{(\xi{\Leftarrow}x)_{b} \;\|}{\sigma \wedge (\xi{:}t)_* \;\|\; \begin{array}{l}\sigma \wedge (t{=}x|t')_a \wedge (\xi{:}t')_* \wedge \\ \mathsf{alive}(t')_a \wedge (t'{=}\mathit{view}())_a\end{array}} \quad \text{if} \begin{cases} t' \text{ is a fresh variable} \\ \sigma \models \mathsf{correct}(\xi, a)_b \\ \sigma \models \mathsf{replicated}(\xi) \\ \sigma \models a{\leftrightarrow}* \end{cases} \tag{7.28}$$

State failure. An important property of cells is that they fail once their state is lost. In other words, if the store σ has no occurrence of a state predicate $\xi{:}z$ anywhere, the cell ξ must fail. In both the stationary and replicated protocols, this situation follows from the failure of the coordinator site of the cell. But in the migratory protocol, we have to add a specific rule that removes the predicate $\mathsf{alive}(\xi)_a$ from the store:

$$\frac{\|}{\sigma \wedge \mathsf{alive}(\xi)_a \;\|\; \sigma} \quad \text{if } \sigma \not\models (\xi{:}z)_b \text{ for all } z \text{ and } b \tag{7.29}$$

7.5 Failure handling

7.5.1 Failure detectors

We provide a generic rule that reflects how the system may update the fault stream of an entity on a site. Upon creation, every entity e in the system has a fault stream on each site a, which is described in the store as the predicate $(\mathsf{fs}(e){=}s|t)_a$, where s is the current fault state of e, and t is the tail of the stream. The latter is a read-only view, but for the sake of simplicity we will treat it as a plain logic variable. The programmer can access it by calling `GetFaultStream`:

$$\frac{(\{\texttt{GetFaultStream}\ e\ x\})_a \;\|\; (x{=}s|t)_a}{\sigma \;\|\; \sigma} \quad \text{if } \sigma \models (\mathsf{fs}(e){=}s|t)_a \tag{7.30}$$

The fault stream of e on a is updated by the rule

$$\frac{\|}{\sigma \wedge (\mathsf{fs}(e){=}s|t)_a \;\|\; \begin{array}{l}\sigma \wedge (\mathsf{fs}(e){=}t)_a \wedge \\ \mathsf{alive}(t')_a \wedge (t{=}s'|t')_a\end{array}} \quad \text{if} \begin{cases} t' \text{ fresh variable} \\ \mathit{cond}(s, s') \end{cases} \tag{7.31}$$

where the condition $cond(s, s')$ is defined below. The site h is the coordination site of e (its home site). Note that s in (7.34) must be different from `permFail`.

$$cond(\texttt{ok}, \texttt{tempFail}) \quad \text{iff} \quad \sigma \not\models \mathsf{alive}(e)_h \quad \text{or} \quad \sigma \not\models a{\leftrightarrow}h \tag{7.32}$$

$$cond(\texttt{tempFail}, \texttt{ok}) \quad \text{iff} \quad \sigma \models \mathsf{alive}(e)_h \quad \text{and} \quad \sigma \models a{\leftrightarrow}h \tag{7.33}$$

$$cond(s, \texttt{permFail}) \quad \text{iff} \quad \sigma \not\models \mathsf{alive}(e)_h \quad \text{and} \quad \sigma \models a{\leftrightarrow}h \tag{7.34}$$

$$cond(s, s') \quad \text{is false otherwise} \tag{7.35}$$

As you can see in the first condition, a temporary failure for an entity e can be reported on a site a when either the entity actually failed, or the network link between a and h is down. The third condition states that detecting the permanent failure of an entity requires site a to be able to reach the home site of the entity. This condition is not fulfilled in general if the site h has crashed. However, it can happen in certain cases, for instance if sites a and h are on the same local area network (LAN). The operating system may report the crash of the process corresponding to site h.

The current fault state of an entity on a given site, as it is defined in Section 7.2 on page 95, is derived from its fault stream on that site. We define it with the following inference rule.

$$\frac{\sigma \models (\mathsf{fs}(e){=}s|t)_a}{\sigma \models (\mathsf{fstate}(e){=}s)_a} \tag{7.36}$$

Fault stream of variables As stated in Section 4.2.3, the fault streams of unified variables are merged. In order to define which one is bound to the other, we assume that all variables in the system are ordered by a relation $\prec$. This order is used by the system, but not directly available to the programmer itself. So the semantic rules below give the possible ways to merge. The last rule finalizes a variable's fault stream when the variable is determined.

$$\frac{\begin{array}{l}\sigma \wedge (\mathsf{fs}(x){=}s|t)_a \\ \wedge\, (\mathsf{fs}(y){=}s'|t')_a\end{array} \;\Big\|\; \begin{array}{c}\sigma \wedge t{=}t' \\ \wedge\, (\mathsf{fs}(y){=}s'|t')_a\end{array}}{} \quad \text{if } \sigma|_a \models x{=}y \wedge x{\prec}y \wedge s{=}s' \tag{7.37}$$

$$\frac{\begin{array}{l}\sigma \wedge (\mathsf{fs}(x){=}s|t)_a \\ \wedge\, (\mathsf{fs}(y){=}s'|t')_a\end{array} \;\Big\|\; \begin{array}{c}\sigma \wedge t{=}s'|t' \\ \wedge\, (\mathsf{fs}(y){=}s'|t')_a\end{array}}{} \quad \text{if } \sigma|_a \models x{=}y \wedge x{\prec}y \wedge s{\neq}s' \tag{7.38}$$

$$\frac{\sigma \wedge (\mathsf{fs}(x){=}s|t)_a \;\|\; \sigma \wedge t{=}\texttt{nil}}{} \quad \text{if } \sigma|_a \models det(x) \tag{7.39}$$

Creation and finalization of the fault stream. Assuming that every site maintains a fault stream for every entity in the system may be misleading. Indeed, this does not take into account the fact that a site may forget some information about an entity (see the discussion of Section 4.4 on page 54). So we propose the following two rules for creating and finalizing the fault stream

of an entity e on a site a. The concept of *liveness* of an entity is the usual one used by garbage collectors. For the sake of conciseness, we skip its formal definition.

$$\frac{\mathcal{T} \quad\|\quad \mathcal{T}}{\sigma \quad\|\quad \sigma \wedge (\mathsf{fs}(e){=}\mathtt{ok}|t)_a} \quad \text{if} \begin{cases} e \text{ is alive on } \mathcal{T}|_a/\sigma|_a \\ \sigma \not\models (\mathsf{fs}(e){=}\ldots)_a \\ t \text{ is a fresh variable} \end{cases} \tag{7.40}$$

$$\frac{\mathcal{T} \quad\|\quad \mathcal{T}}{\sigma \wedge (\mathsf{fs}(e){=}s|t)_a \quad\|\quad \sigma \wedge t{=}\mathtt{nil}} \quad \text{if } e \text{ is not alive in } \mathcal{T}|_a/\sigma|_a \tag{7.41}$$

7.5.2 Making entities fail

In this section, we define the operations `Kill` and `Break`.

Global failure. The operation `Kill` should make its argument fail, i.e., it should remove the predicate $\mathsf{alive}(x)_a$ from the store, where x is the argument of the call. As the operation is asynchronous, we use a "kill" message $x{\Leftarrow}\dagger$ that is similar to the messages used in Section 7.4.2.

$$\frac{(\{\mathtt{Kill}\ x\}\ T)_b \quad\|\quad T_b, (x{\Leftarrow}\dagger)_b}{\sigma \quad\|\quad \sigma} \tag{7.42}$$

$$\frac{(x{\Leftarrow}\dagger)_b \quad\|\quad}{\sigma \wedge \mathsf{alive}(x)_a \quad\|\quad \sigma} \quad \text{if } \sigma \models a{\leftrightarrow}b \tag{7.43}$$

Notice that the latter rule states explicitly that communication with the coordinator site of x is necessary to make x permanently failed.

Local failure. The procedure `Break` is pretty easy to define. Its effect is to change the fault state of the entity to `localFail`, unless the fault state already has that value or `permFail`.

$$\frac{(\{\mathtt{Break}\ x\})_a \quad\|\quad (\mathbf{skip})_a}{\sigma \wedge (\mathsf{fs}(e){=}s|t)_a \quad\|\quad \begin{array}{c}\sigma \wedge (\mathsf{fs}(e){=}t)_a \wedge \mathsf{alive}(t')_a \\ \wedge\ (t{=}\mathtt{localFail}|t')_a\end{array}} \quad \begin{array}{c}\text{if } \sigma \models s{=}\mathtt{ok} \text{ or} \\ \sigma \models s{=}\mathtt{tempFail}\end{array} \tag{7.44}$$

$$\frac{(\{\mathtt{Break}\ x\})_a \quad\|\quad (\mathbf{skip})_a}{\sigma \wedge (\mathsf{fs}(e){=}s|t)_a \quad\|\quad \sigma \wedge (\mathsf{fs}(e){=}s|t)_a} \quad \begin{array}{c}\text{if } \sigma \models s{=}\mathtt{localFail} \\ \text{or } \sigma \models s{=}\mathtt{permFail}\end{array} \tag{7.45}$$

If site b has locally killed an entity e, the predicate $\mathsf{correct}(e, a)_b$ will never be entailed by the store. Hence, all operations that require e to be correct on b block forever.

7.6 Mapping distributed to centralized configurations

The definition we gave of a refinement on page 93 states that every distributed configuration maps to a centralized configuration. This section defines precisely that mapping.

7.6.1 The mapping

The mapping itself is pretty easy to define. Basically we collect the configurations of all sites in the system. We define it in terms of the operator $|_a$. The disjoint union and conjunction operators range on the set of all sites in the system.

$$\mathit{centralized}\left(\frac{\mathcal{T}}{\sigma}\right) = \frac{\uplus_a \mathcal{T}|_a}{\wedge_a \sigma|_a} \tag{7.46}$$

There is only a small issue with the conjunction of the local stores. They are always consistent with each other, except for the entities' fault streams, which can be in different states. However, we can consider that a centralized configuration possibly has several fault streams for a given entity. This looks like the centralized system maintains many failure detectors for each entity, which can have different views.

7.6.2 Network transparency

With this definition we can now formulate a theorem which relates the distributed and centralized semantics of the language Oz. As this property translates the network transparency at the semantic level, we call it the *Network transparency theorem*. The theorem can be proven by induction on every distributed reduction rule.

Theorem (Network transparency)**.** The distributed semantics of the language is a refinement of its centralized semantics. In other words, for every pair of distributed configurations $\mathcal{D}$ and $\mathcal{D}'$, if $\mathcal{D} \to \mathcal{D}'$ is a valid distributed reduction, then $\mathit{centralized}(\mathcal{D}) \to \mathit{centralized}(\mathcal{D}')$ is a valid centralized reduction.

8

Implementation

This chapter describes the new implementation of the distribution of Oz, based on the Distribution Subsystem (DSS). This work has been achieved by several developers, among which Erik Klintskog, Zacharias El Banna, Boris Mejías, and myself. In order to make a clear distinction between the former and new implementations, we will refer to them as "Mozart" and "Mozart/DSS", respectively.

In Section 8.1, we explain the architecture, and some principles underlying Mozart/DSS. We show there that the implementation splits up into three distinct layers. The topmost layer is the virtual machine, which has been modified as little as possible in order to take distribution into account. Section 8.2 describes the bottom layer, which provides all the distributed protocols. Section 8.3 describes the middle layer, also known as the Glue, that interfaces the virtual machine to the DSS layer.

8.1 Architecture of Mozart/DSS

The platform Mozart/DSS is a new implementation of the distribution of Oz, based on the virtual machine of Mozart and the library DSS. The latter provides abstractions for distributing programming language entities. The general architecture for a distributed entity is depicted in Figure 8.1 on the following page. The diagrams shows the fundamental components implementing an entity that is shared among three sites. The components are divided up into sites, separated by bold vertical lines, and into implementation layers, separated by dashed horizontal lines.

All language entities, local or distributed, are stored in virtual machine heaps. A distributed entity can be seen as a set of local entities connected together via the network, and cooperating in order to provide the illusion of a single global entity to the programmer. Note that an entity in the heap might

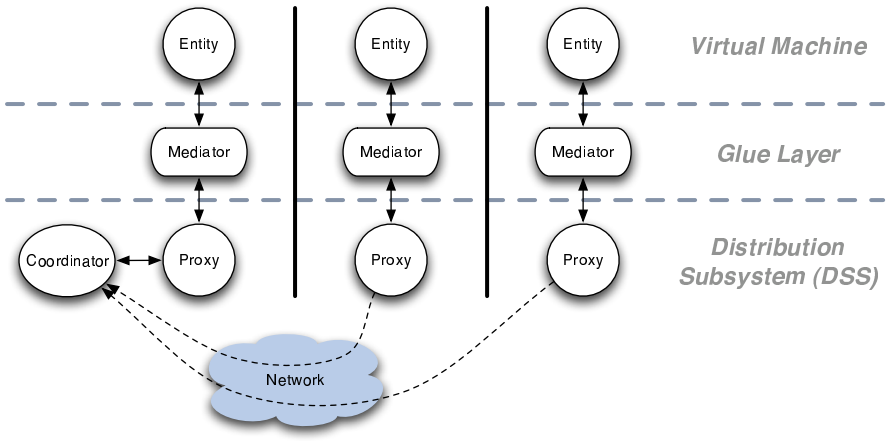


Figure 8.1: The three layers that implement the distribution in Mozart/DSS

be only a *stub*, i.e., the state of the entity is not available locally. In that case, it is necessary for that site to cooperate with other sites in order to complete a language operation.

A distributed entity has a special hook that connects it to a *proxy* in the DSS library. The proxies of a given entity are connected together with a *coordinator* via network links. The coordinator and proxies form the *coordination network* of the entity, which has a unique global identity. It is used to identify the language entity across sites boundaries. The coordination network implements the *access architecture* of the entity, which we introduced in Section 3.3.4.

The role of each layer. Each layer in the implementation plays a specific role. The virtual machine layer implements the entity's centralized semantics. The DSS layer provides global naming for entities, a general serialization mechanism for user data, a set of selectable protocols implementing generic entity operations, a distributed garbage collector with several protocols available, and failure detectors for sites and entities.

The Glue layer implements the distributed semantics of entity operations by mapping them to DSS entity operations. It implements the failure semantics of entities, and makes both the local and distributed garbage collectors cooperate. It also provides network communication channels for the DSS.

The author's contributions. A prototype of the Glue layer was given to us by Erik Klintskog. We revised its design, and implemented it entirely, with a little help from our colleage Boris Mejías. Together with Boris we modified the Mozart marshaler to serialize and deserialize entities using the DSS. The new language features, like annotations, the fault stream, and the operations

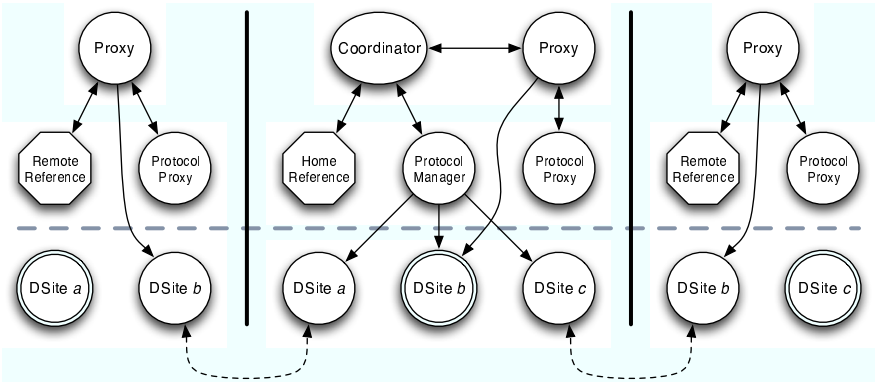


Figure 8.2: Architecture of the DSS, shown for one entity shared among three sites a, b, and c

`Kill` and `Break`, were implemented solely by the author. We rewrote all the entity protocols in the DSS such that they could handle partial failures. We also extended the DSS interface to handle and reflect entity failures.

We made a few contributions to the virtual machine, too. Together with Fred Spiessens, we implemented our new design of the by-need synchronization mechanism. We also had to adapt the virtual machine in order to handle the new distribution of procedures and object, among others. Finally, we improved the unification's implementation to make it more incremental.

8.2 The Distribution Subsystem

The DSS library provides a set of protocols for distributing programming language entities [Kli05]. It is itself split into a *protocol layer*, and a *messaging layer*. The protocol layer brings together all the protocols to manage distributed entities. Protocols are partitioned in three classes, that handle orthogonal aspects of an entity's distribution.

- *Coordination protocols* provide an entity's unique identity, and maintain its coordination network. They let proxies and their coordinators reach each other by message passing.
- *Entity protocols*, or *consistency protocols*, implement generic entity operations. They also handle partial failures of entities.
- *Reference protocols* implement distributed garbage collection policies for shared language entities.

Figure 8.2 shows the main DSS components for one entity shared among three sites a, b, and c. The sites are separated by the vertical bold lines, and

the horizontal dashed gray line splits up both layers. One can see that each site has a proxy for the entity, and site b owns its coordinator; those components implement the coordination protocol of the entity. Each proxy is connected to a *protocol proxy*, while the coordinator has a *protocol manager*; protocol proxies and manager implement the consistency protocol of the entity. Finally, the coordinator owns a *home reference*, and the remote proxies have a *remote reference*; those components implement the distributed garbage collection protocol of the entity.

Some of those components, like a proxy with its protocol proxy, may call each other directly, but they interact more generally via the messaging layer. The latter provides a channel-based communication mechanism (reliable and ordered message passing) between sites. Each site is abstracted by a *DSite* component, which hides the communication channel, and reflects the fault status of the given site. The DSS on each site maintains a set of known sites, as it is shown in Figure 8.2 on the preceding page. Each site also has a representation for itself, drawn with double lines in the figure.

Each component in the protocol layer can be addressed with its type (proxy, coordinator, protocol proxy, protocol manager, or reference), the global identity of its coordination network, and a DSite. For instance, the protocol proxy on site c can easily send a message to its protocol manager via its proxy, which knows the global identity and the DSite of its coordinator. The protocol manager on site b can send a message to its protocol proxy on site a, with its own reference to the DSite of a, and the global identity of its coordinator. This facility greatly simplifies the implementation of the protocols.

By design, the coordination network provides a mean for each proxy to send messages to its coordinator. In case the coordinator is stationary, each proxy just has to know the coordinator's site. All the DSS protocols are built on top of this architecture. By default the protocol manager does not know its proxies, so in some cases it maintains a list of DSite references corresponding to its proxies. This list is built either explicitly by making proxies register to their manager, or implicitly by collecting message origins. The latter case uses the fact that a message is always delivered together with the DSite representing the sender's site.

8.2.1 Protocols for mutables

Those protocols implement three operations, namely *read*, *write*, and *send*. The operations read and write are synchronous, and may return a result. The send operation is similar to an asynchronous version of a write, and does not return any value. There are essentially four protocols available in this category: the stationary state protocol, the migratory state protocol, the pilgrim protocol, and the invalidation protocol. The author made significant contributions to the last three ones.

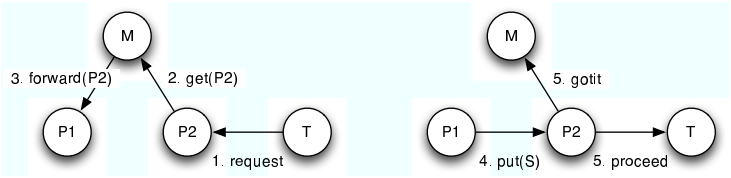


Figure 8.3: Basic migratory state protocol

The stationary state protocol

This protocol is the simplest of all. Remote proxies send their operation requests to their protocol manager, which performs the operation. If the operation is not a send, it returns a result message once the operation on the entity has completed. The result message allows the proxy to resume the corresponding suspended operation on its site.

The migratory state protocol

This protocol was first described in [HVS97, VHB+97, HVBS98], then extended in [VBHC99] to make it handle permanent failures. The author proposed a formalization of the extended protocol, and proved it correct in [BVCK00]. Mozart used that protocol for distributed cells and objects.

The protocol uses a token which is passed between the proxies in the coordination network. The proxy holding the token has sole access to the state of the distributed language entity, and the entity's state is passed together with the token. The migration of the token is shown in Figure 8.3. The protocol manager M builds a forwarding chain with all the proxies requesting an operation. When a proxy P2 needs the state of the entity to perform an operation for thread T, it sends a message `get(P2)` to M. The latter then sends a message `forward(P2)` to the last proxy in the forwarding chain. This message will make P1 forward the state token to P2, so that P2 becomes the last proxy in the forwarding chain. When P2 receives the state token, it sends a message `gotit` to its manager. This message allows M to maintain a list of the proxies that could hold the state token.

Bypassing failed proxies. This simple extension to the basic protocol allows a proxy to avoid sending the state token to a failed proxy. This situation is depicted in Figure 8.4. The proxy P1 has detected that its successor P2 in the chain is permanently failed. In order to find the next successor, it notifies its manager. The manager M can then send a new message `forward()`, because M owns a representation of the forwarding chain.

State loss detection. The state token may be lost either if a proxy holds it and crashes, or it has been sent over the network in a message `put`, and the

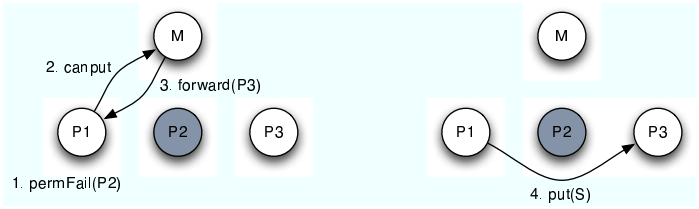


Figure 8.4: Bypassing a failed proxy

message is lost because of a site failure (of the sender or the receiver). When the manager detects that a proxy in the chain has permanently failed, it runs an inquiry protocol, which determines whether the state token has been lost. The manager asks each proxy where the state token is. The proxy can answer `beforeMe`, `atMe`, or `afterMe`. If the manager finds two proxies that answer `afterMe` and `beforeMe`, all proxies between them have crashed, and there is nothing in the network, then the entity state is lost. The permanent failure of the entity is notified to all proxies.

The pilgrim protocol

This mobile state protocol is inspired by the work in [GLT97]. It can be seen as a variant of the migratory token protocol, where the proxies accessing the token form a ring instead of a chain. Each proxy in the ring has a successor, to which it forwards the token. A proxy remains in the ring unless it has not performed any entity operation for a certain period of time. The proxies interact with the manager only to enter or leave the ring. This greatly reduces the interaction with the manager when a set of proxies regularly access the token.

The author made a significant contribution to make that mobile state protocol handle failures. The original implementation, as provided by [Kli05], had no simple way to detect whether the state token was lost. Moreover, proxy insertions and removals in the ring were serialized at the manager. This implied a strong protocol invariant which was relied upon for garbage collection, because it allowed proxies to know whether they were inside the ring, and consequently whether they had to be kept alive. Proxies inside the ring should not be removed by their respective garbage collectors, since their removal would create a gap in which the state token can be lost. But determining efficiently when a proxy is no longer accessible from the ring can be tricky, in particular when ring proxies crash.

In order to remove any dependency of the manager on its proxies, we have simplified the proxy insertion and removal in the ring. The result is shown in Figure 8.5. Dashed arrows represent the successor relation in the ring. When the manager receives a request from a proxy P to enter the ring, it chooses two consecutive proxies P1 and P2 in the ring. It then sends a message to P1 to make its successor P, and another message to P to make its successor P2, so

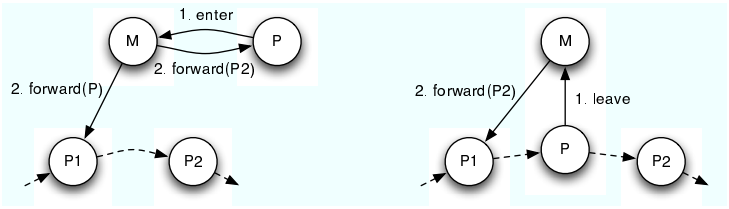


Figure 8.5: Pilgrim: entering and leaving the ring

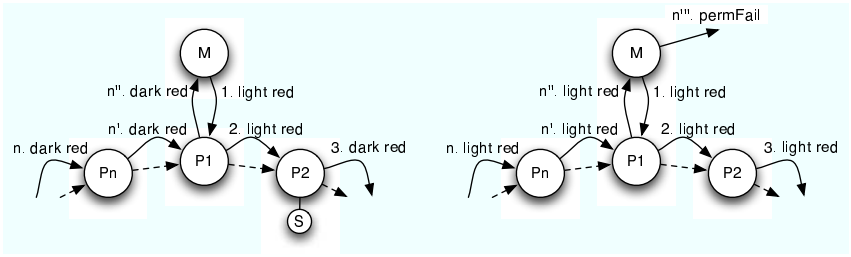


Figure 8.6: Pilgrim: ring coloring

that P is eventually between P1 and P2 in the ring. Proxy removal is even simpler: its ring predecessor is sent a new successor. The simplified protocol allows the manager to remove any suspect proxy from the ring without that proxy's cooperation. To compensate for the apparent sloppiness of the protocol, we have added an orthogonal protocol that both detects the loss of the state token, and solves the garbage collection issue.

Ring coloring. The idea of the coloring protocol is to mark the proxies that are inside the ring, following the ring structure. The proxies that have not been marked during the process are guaranteed to be unreachable from the ring, and can therefore be safely removed. Moreover, the protocol is able to detect whether the coloring token or a newly marked proxy has encountered the state token. At the end of the coloring, the manager checks this information, and notifies all proxies if the state token is no longer present.

More than two colors are necessary to make the protocol robust. The coloring can be interrupted at any moment by a failed proxy, and several color changes can be performed concurrently. Eventually all proxies will change to the most recent color.

Figure 8.6 shows how the coloring protocol works. The protocol attaches a color to every proxy and the state token, and the color is either "light" or "dark". A proxy always attaches its own color to the state token. When the manager initiates a color change, it sends a message to one of the proxies in the ring with a light color (red in the figure). The proxy creates a color token, and

passes it around the ring. The color token changes the color of every proxy it encounters. If the color token meets the state token, its color is darkened, and the coloring continues. The first proxy also darkens its color when it receives the state token. When that proxy receives the color token, it knows that the state token has been lost if and only if the token's color and its own color are equal and light. The final color is sent back to the manager. Note that the proxies never accept a state token with a less recent color, except the proxy that initiated the coloring.

A color change is triggered each time a proxy fails, or when a proxy wants to determine whether it is still reachable from the ring. The proxy is guaranteed to be unreachable if its color has not been changed by the process. The manager keeps track of the proxies that left the ring, and forwards them the new color after the coloring. A proxy that has a different color knows that it is unreachable from the ring.

The invalidation protocol

This protocol, inspired by protocols presented in [Lam79], manages an entity whose state is replicated on its proxies. It implements the annotation `replicated`. It maintains two types of tokens, multiple read tokens and a single write token. Holding a read token allows a proxy to read its local copy of the state of the entity. To perform a write operation, the manager asks proxies to release their read token, and *invalidate* their copy of the state. Once all read tokens have been collected, the write token is used to update the state, then read tokens are redistributed to proxies with the new state. The proxies delay read operations until they receive a read token.

In its first formulation, the manager was giving the write token to the proxy that requested the write operation [Kli05]. The new state was then sent to the manager, which redistributed it with read tokens. However, that protocol was sensitive to proxy failures. The author has chosen to simplify its failure modes by performing all write operations on the protocol manager. This makes the protocol insensitive to proxy failures. It also improves performances, since the manager no longer has to send the write token to a proxy.

8.2.2 Protocols for immutables

Those protocols should only offer a *read* operation. The stationary protocol is valid for immutable entities, as well as three others, namely the immediate protocol, and the eager and lazy replication protocols. Their implementation in the DSS was done by Per Sahlin [Sah04], then slightly extended by the author to handle partial failures.

The immediate protocol is not really a protocol, since a full representation of the entity is serialized when a reference is passed between sites. The eager and lazy replication protocols are pretty simple: each proxy can ask its manager a copy of the entity's state. Once the state is installed, all read operations are

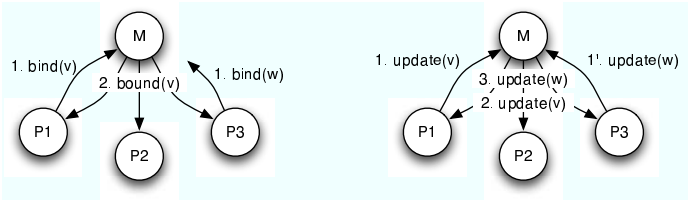


Figure 8.7: Transient protocol: bind and update operations

performed locally. In the eager protocol, a proxy requests the state right after its creation. In the lazy protocol, the proxy delays the state request until the first read operation. Both the eager and lazy protocols guarantee a unique copy of the entity's state. Indeed, if a reference to the entity is sent to a site, that site will identify the reference to the entity's proxy. If the proxy had already requested the entity state, it will not request it again.

8.2.3 Protocols for transients

This protocol implements single assignment variables, with two operations: *bind* and *update*. The protocol has first been published in [HVB+99]. The DSS implements this protocol extended with an incremental update operation. Upon creation, the transient entity is unbound. Its transient state may be updated as many times as desired, until the entity is bound. The binding is unique and final; all subsequent updates and bindings will fail. By-need synchronization in Mozart/DSS is implemented as an update (see Section 8.3.4).

Bind. In order to guarantee the unicity of the binding, the protocol manager plays the role of an arbiter of binding attempts. The protocol is depicted in Figure 8.7: proxies send binding requests to their manager; the latter accepts the first request, and forwards the binding to all proxies. All subsequent binding and update requests are ignored. When proxies receive the binding, they install the final state in their entity, and check their former binding attempts to decide whether they have succeeded.

As the manager broadcasts the binding to all its proxies, they must register to their manager, unless the coordination network provides a way to reach them all. The registration is done at proxy creation, when it is deserialized, via an explicit registration message sent to the manager. If the entity is bound at the message reception, the manager replies with the entity's binding. Note that the registration can be optimized if the entity reference was sent by the home proxy: the manager may automatically register the destination site. This *autoregistration* mechanism saves a registration message, and can improve the throughput of the protocol in case of stream communication.

Update. State updates are handled in a similar way, but they do not put an end to the entity. All updates are serialized by the manager, which forwards them to all known proxies, so that the updates are applied in the same order on all proxies (see the right drawing in Figure 8.7). When a proxy registers, the manager may send back an update that *summarizes* all former updates. This guarantees that the proxy does not miss past updates, provided that the entity's state can reflect all past updates. This summary update is a contribution of the author. Its absence creates a race condition between proxy registration and updates.

The transient remote protocol. This variant of the transient protocol, chosen by the annotation `reply`, delegates the arbiter role to a proxy. That proxy can directly bind the entity, and forward the binding to the manager, which broadcasts it to the other known proxies. The manager forwards all the other updates and binding requests to that proxy, which serializes them. The protocol is optimal if there is only one remote proxy, and that proxy binds the entity. Indeed, the proxy is autoregistered because the entity reference must have been sent from the manager's site, and the only message actually sent is the forwarded binding.

The manager is responsible for choosing the arbiter proxy. The simplest rule is to choose the first proxy registered outside the manager's site. If that proxy is deleted, it sends a deregistration message to the manager, which reassigns the arbiter role to its home proxy.

8.2.4 Handling failures

All entities supported by the DSS may fail. Failures have two origins: the environment and the programmer, and both kinds must be detected and reflected to the user. To implement that, each coordination proxy maintains a failure state for its entity. Each time that state changes, the proxy notifies its corresponding mediator in the Glue layer. The state may be changed by the proxy itself, or its protocol proxy.

Failures due to the environment are detected via DSites. We take for granted that DSites have their own failure detection mechanism, which reflects their corresponding site's fault state. Consider a DSite representing site b on a site a. Once the DSite changes its fault state, this change is notified to all coordination and protocol components on site a. Every component checks whether it affects its entity. If so, it changes its failure state accordingly, and notifies the mediator of its entity. How an entity is affected depends on the entity's coordination architecture and protocol. For instance, the mobile state protocols will probe the proxies that may hold the entity's state, in case one of those proxies is reported as failed.

Failure are not only reported locally, but also to a global scale. A generic protocol supports this global reporting. Once an entity is diagnosed as permanently failed by a protocol manager, the latter broadcasts a message `PERMFAIL`

to all its known proxies. Those proxies will then update their failure state, and report the change to their own mediator. Note that this generic protocol is sometimes adapted to avoid inconsistencies. For instance, the transient protocol never makes an entity permanently failed after its binding.

Kill. In order to let a program make an entity fail, all protocols support an operation called *kill*. The same generic protocol is used to implement that operation. To perform a kill, a protocol proxy simply sends a message `PERMFAIL` to its manager, which propagates the failure globally as we explained in the former paragraph.

8.2.5 Distributed garbage collection

The garbage collector provided by the DSS maintains a status for each coordination proxy in the system. The proxies of a given entity have different status, depending on the references and the entity's protocol. A given proxy can be in one out of four states:

- `PRIMARY`: the entity is kept alive by remote references. The virtual machine must keep the entity, because other sites depend on it. This status means that the coordinator of the entity is on the current site.
- `WEAK`: the entity is kept alive because for protocol needs. This typically happens when the current site is the only one to hold the entity's state. That status is generally not definitive: the DSS can be instructed to move away from that status.
- `LOCALIZE`: no remote reference keeps this entity alive. This usually means that all the proxies of the entity are gone except this one. The entity can be localized or deleted, depending on whether it is kept alive locally.
- `NONE`: the liveness of the entity entirely depends on local information. If the entity is alive, the proxy should be kept. Otherwise, both can be removed.

Note that proxies are considered alive by the DSS until they are deleted explicitly by the upper layer. In fact all the components at the interface of the DSS library must be deleted explicitly. The DSS uses them as roots for its own internal garbage collector.

Section 8.3.6 on page 123 explains how these states are used by the local garbage collectors. The distributed garbage collector itself is implemented by the *reference* components shown in Figure 8.2 on page 109. Several protocols are available, and they can be combined together. The most important protocols are a weighted reference counting algorithm, and a time lease algorithm. More details on the algorithms are given in Erik Klintskog's works [Kli05, KNBH01].

8.3 The language interface

The Glue layer implements the language interface to the distribution library. As the distribution of Oz comes from sharing entities, the most important component of this layer is the *mediator*, that interfaces an entity to its proxy, and vice-versa. Every distributable entity has a mediator, which contains the entity's annotations, its fault state, and its memory status. For the sake of performance, the mediator of a local entity is created lazily, once the entity is annotated, broken, or serialized for a remote site.

An entity is distributed if and only if it has a proxy in the DSS layer, otherwise it is purely local to its site. In general, distributed entities have a pointer to their mediator, while local entities have an indirect access to their mediator via a table. The mediator itself has pointers to both its entity and proxy, and the latter has a pointer to the mediator. This provides both efficient access for distributed entities, and small overhead for purely local entities.

8.3.1 Distributed operations in general

Performing a language operation on a distributed entity involves several components in the three layers of Mozart/DSS. Those components are depicted in Figure 8.8 on the next page, with the horizontal dashed lines separating the implementation layers. Each one has a specific role during the distributed execution of the operation. Note that entity failures are ignored here; they are explained in the next section.

A language operation on a distributed entity is always delegated to the Glue layer, which accesses the entity's mediator, then the corresponding DSS proxy, and invokes the latter with a generic operation. The proxy forwards the call to its protocol proxy, which implements the protocol chosen for the entity. The protocol proxy prescribes to either perform the operation locally, as if the entity was purely local, or suspend and resume it later. The decision entirely depends on the protocol.

Let us illustrate the decision taken in the case of the protocol `migratory` (mobile state). If the entity's state is on the site that attempts the operation, it will be performed locally. This is fine, since the state is stored in the virtual machine's representation of the entity. If the entity's state is not present, the operation is suspended, and the distributed protocol is used to complete the operation. Once performed, it is resumed, together with the thread that attempted it.

Suspension and resumption. If the operation has to be suspended, the Glue creates an object that represents the suspended operation. That object must be able to resume the operation whenever the protocol says so. Resumption may be done in two ways: either

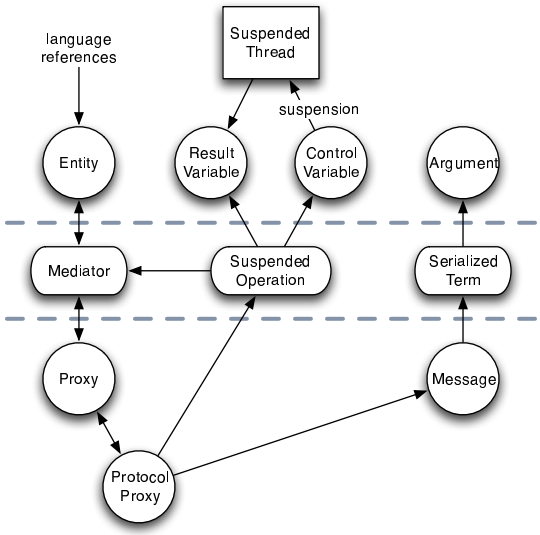


Figure 8.8: The components involved into the distributed execution of a language operation

- the protocol installs a copy of the entity's state in the local entity, and the operation is performed locally, or

- the operation has been performed remotely, and the operation resumes by delivering the results.

The suspended operation typically suspends the thread that attempted the operation on a control variable. Once the operation completes, the thread is woken up by binding the variable. The control variable also permits to raise an exception in the thread, or resume the operation with another statement. The suspended operation can also deliver an output from a remote execution through a result variable.

When an operation is performed on a remote site, the entity's protocol proxy on that site invokes the corresponding mediator in order to perform a virtual machine operation.

Passing values. Protocol messages may include Oz values. Those values can be the input or output of an operation that is performed remotely, or a value that represents the entity's state. They are encapsulated in a Glue component that takes care of their (de)serialization. The DSS library defines an interface for a suspendable marshaler. It means that values are generally not serialized as a whole, but the serialization is done until a buffer is almost full. This technique generally results in a smaller memory footprint.

8.3.2 Distributed immutables

The role of an immutable entity's proxy is to either provide a copy of the entity's contents (protocols `immediate`, `eager`, and `lazy`), or to provide remote access to the entity (protocol `stationary`). We call the first kind of entities *copiable*, because their distribution protocol consists in copying their contents between sites.

Copiable entities do not always require to query the Glue layer to perform an operation. Indeed, once the contents of the entity is available on a given site, all operations on that site will be performed locally. The overhead of distribution can be reduced to nothing if the virtual machine performs this optimization.

Another aspect of copiable entities is that they can survive the execution of a program: they can be stored in a file, and reused later, possibly by another program. This can be an issue if the entity's identity is provided by its coordination network, because the latter is no longer functional once the program stops. For this reason, we provided the entity with a global identifier that does not refer to a live DSS component. As a consequence, it is possible to remove the entity's proxy once it has been copied. This can be done for instance by the garbage collector.

8.3.3 Remote invocations and thread migration

Stationary objects and procedures are never copied between sites, only their reference is transmitted between sites. The operation they have in common is the call (also called invocation in the case of objects). For this operation, an object can be seen as a special case of a procedure. Therefore the discussion will only mention procedures, and calls to stationary procedures.

Assume that a thread on a site a attempts to execute the call statement `{`$p\ x_1 \ldots x_n$`}`, where p is a stationary procedure on a site b. If the protocol proxy of p has to perform the call remotely, it asks the Glue layer to transmit arguments for the remote operation, i.e., in our case x_1, ..., x_n. The Glue layer on site b then simply calls p again with the given arguments on a new thread, which will execute p because it is now on its site. As the procedure call may exit with an exception, site b creates a variable z, and pushes the following statement on the thread:

```
z = try {p x1 ... xn} unit
    catch E then {FailedValue E} end
```

The variable z is returned to site a, which replaces the original call to p by `{Wait` z`}`. This automatically synchronizes the thread, and transmits an exception if needed. This is illustrated in Figure 8.9 on the facing page, for a procedure `P` with two arguments.

This simple solution works in most cases, but it has a subtle issue: the calling thread, and the one that actually executes the procedure have different

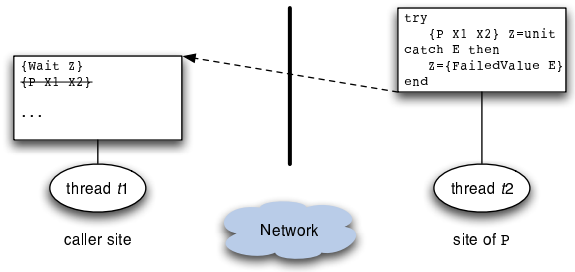


Figure 8.9: A remote procedure call

thread identifiers. This is a problem if these threads lock critical sections with *reentrant locks*. Those locks allow a thread to enter many times in a critical section, and use the thread identifier to distinguish between threads. For the remote procedure call to be *really* transparent, both threads should have the same identifier, just like if the thread has *migrated* between sites. Otherwise, a deadlock may occur. In the example of Figure 8.9, we should have $t1 = t2$.

To realize this, the identifier t of the caller thread is sent together with the procedure's arguments. On site b, the procedure is executed on a thread with identifier t. If no such thread exists on b, one is created. If such a thread exists, then by design it must be suspended on another remote procedure call; the topmost statement must be a call to `Wait` as above. Pushing a new statement on that thread is safe, because after that statement is reduced, the thread will re-suspend thanks to the `Wait` statement.

8.3.4 Unification and by-need synchronization

While the unification operation belongs to the virtual machine, its implementation deserves a special attention. The reason is that a single unification may involve several concurrent distributed variable bindings. Performing those distributed bindings sequentially may have a significant impact on the operation's performance. This impact may be reduced to a minimum if those bindings are truly performed in parallel. The author has modified the existing implementation of the unification in Mozart, which was interrupted by every distributed binding it had to perform.

The unification of two terms basically traverses two directed graphs, and binds the encountered variable nodes in order to make both graphs equivalent. If a variable is local to a site, its binding is done immediately. But a distributed variable may require to invoke its protocol. In that case, the binding is suspended until the protocol terminates the operation, and returns its result. Suspending the whole unification at that point is correct but inefficient, since all involved distributed bindings will be performed in sequence.

To make a better implementation of the unification's semantics, the original

algorithm is modified as follows. When a variable binding suspends, it is put aside in a "suspended set". Note that bindings of read-only variables are also put in that set. The bindings in the suspended set are considered valid until the algorithm terminates. If the algorithm terminates without failing, and the suspended set is nonempty, say $\{x_1{=}v_1, \ldots, x_n{=}v_n\}$ with $n > 0$, the unification resumes as unifying two tuples with the remaining bindings, i.e.,

$$x_1\#\ldots\#x_n = v_1\#\ldots\#v_n.$$

Moreover, the current thread is suspended on the variables x_1, ..., x_n. The thread will be woken up as soon as one of those variables is bound (possibly by a distributed binding), and the unification will make progress. If all the variables x_i are distributed, then all the bindings $x_i{=}v_i$ will proceed concurrently.

Unifying distributed variables. The unification of two distributed variables requires a tiebreaker to decide which variable is bound to the other. This is necessary for avoiding "binding cycles" in the system. Indeed, if two sites attempt to perform $x{=}y$ and decide differently, x may be bound to y and vice-versa. This is problematic, since both transients are bound, and can therefore no longer be bound to anything else: the variables are unbound forever. The tiebreaker is borrowed from the DSS, which provides an arbitrary total order between all its distributed entities. The order is guaranteed to be the same on all sites.

By-need synchronization. As we have seen in Chapter 7, the semantics of by-need evaluation simply requires to propagate the need of a variable on all sites. The operation *update* provided by the transient protocols perfectly fulfills the job. Once a variable is made needed, an update operation is performed, which will make all representatives of that variable needed. Note that making a variable needed never blocks, since that update can be considered asynchronous.

8.3.5 Fault stream and annotations

The mediator of a language entity manages most aspects of to the distribution of that entity. Some of those aspects, like the entity's fault stream, are visible as language entities. Others, like the entity's annotations, are stored directly in the mediator. Figure 8.10 on the facing page shows the extra language entities used by an entity's mediator.

Fault stream and blocking threads. First, the mediator keeps track of the entity's fault state. It is updated whenever a new fault state is reported by the entity's proxy, or enforced by the user (`localFail`). The mediator manages the entity's fault stream by keeping a reference to its tail, a read-only variable. The stream is extended each time the fault state changes. The mediator also

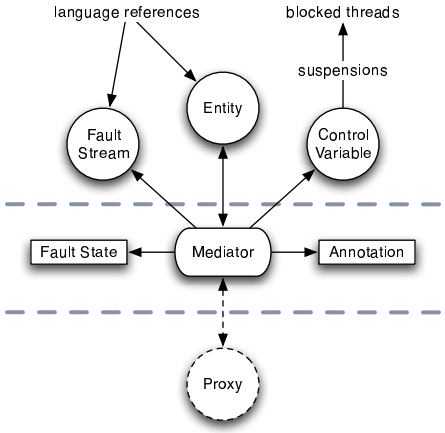


Figure 8.10: The entities managed by an entity's mediator

manages a control variable. If a thread attempts an operation on the entity while its fault state is not `ok`, the thread suspends on that variable. Whenever the fault state becomes `ok`, that control variable is bound to **`unit`**; this automatically wakes up all blocked threads, which will retry their operation.

Note that the memory footprint of the entity pretty small in practice. Both the fault stream and the control variable are created lazily, once the program needs them. Another optimization comes from the fact that the control variable is only effective with the fault state `tempFail`. Indeed, a thread blocking because of the fault states `localFail` and `permFail` will never resume. Therefore the control variable does not need to be kept alive by the mediator in that case, since it will never be bound.

Annotations and proxy. The entity's annotations are also stored in the mediator. They are used to create a DSS proxy for the entity. The creation and removal of the proxy are both managed by the mediator. The proxy creation (called entity *globalization*) is triggered when the entity is serialized, while its removal (called entity *localization*) is prescribed by the garbage collector when the entity is no longer referenced outside its original site.

8.3.6 Garbage collection

The memory management of Mozart/DSS involves both the virtual machine's garbage collector and the DSS's distributed garbage collector. We will call them the local and DSS garbage collectors, respectively. The latter has already been described in Section 8.2.5 on page 117. This section focuses on the cooperation between the implementation layers.

The basic principle is that a live entity keeps its mediator alive, and a live

mediator keeps all its entities (the entity, its fault stream and control variable) alive. We combine that principle with the information coming from both the virtual machine and the DSS. When the Glue layer decides to remove or localize a distributed entity, it deletes the entity's proxy. The following paragraph explains how the decision is taken.

Putting it all together. The cooperation between the garbage collectors is quite generic. First, both the virtual machine and the DSS should provide correct information about what must be kept in memory. Second, some decisions, like the correct handling of the `WEAK` state above, depend on whether an entity is kept alive by local computations only. Because some of those entities have to be kept anyway, the process requires two passes of the local garbage collector. The main steps of the garbage collection process are the following.

1. Distributed entities in state `PRIMARY` are taken as roots for the local garbage collector.
2. The local garbage collector is run, which recursively marks entities from the roots. We can now determine which entities are marked by local computations.
3. The distributed entities in state `WEAK` are checked. For each such entity, if it has not been marked yet, mark it and instruct its proxy to move away from that state.
4. Local garbage collection is performed again. Now all entities that must be kept in memory are marked.
5. The distributed entities in state `NONE` are kept only if they are marked locally; otherwise they are deleted together with their mediator and proxy. The distributed entities in state `LOCALIZE` are localized (their proxy is removed) if they are marked locally; otherwise they are deleted together with their mediator and proxy.
6. The DSS performs its own internal garbage collection. This has no effect at all on the virtual machine's memory heap.

These steps give the broad idea for collecting the entities that must be kept in memory. However some important details are missing, in particular at step 1. The rest of the section identifies the missing roots for the local garbage collection, with a detailed explanation for each. We analyze (in order) the cases of distributed variables, fault streams, threads blocked on a failure, and the components involved in a distributed operation.

Distributed variables. The process described above does the right thing for most distributed entities. However, let us analyze what happens to threads that suspend on distributed variables. Recall that when a variable is alive, its

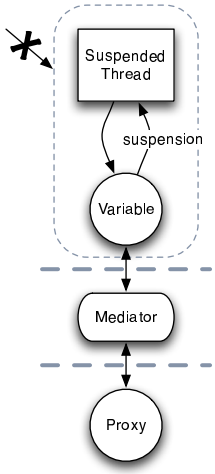


Figure 8.11: A distributed variable with suspensions only

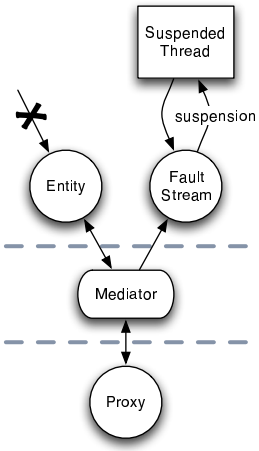


Figure 8.12: A distributed entity with monitoring threads only

suspensions are kept alive. If no live entity refers to any of them, the variable and its suspensions are considered dead. The following situation, depicted in Figure 8.11, might be problematic: a thread suspends on a distributed variable, on a site that does not hold the variable's coordinator. If nothing else keeps that variable alive, it might be considered dead, together with its associated suspended threads. Should the Glue layer keep this variable alive?

Our answer is: yes, distributed variables with local suspensions should be considered as roots for the local garbage collector. The reason is pretty simple. Suspending on a distributed variable is a common idiom, where one site waits for the result of a computation performed on another site. The programmer rarely considers the blocked thread as possibly dead. On the contrary: that thread is often used to keep the continuation of a local computation alive. Silently removing the variable and its suspensions would be an error.

Note, however, that the garbage collector is unable to find out whether there exists a thread in the system that can bind the variable. If no thread binds the variable, all suspensions are dead. The suspensions can also be considered dead if the variable is permanently failed, locally (`localFail`) or globally (`permFail`). If the program is able to detect a dead variable, it can help the local garbage collectors by making the variable fail. We extend the step 1 above with:

1.1. Distributed variables with local suspensions are taken as roots for the local garbage collector, unless they are permanently failed.

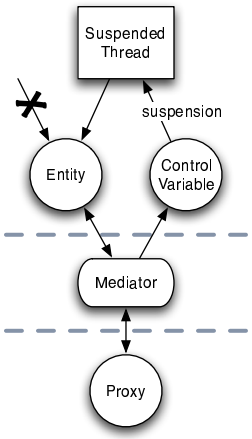


Figure 8.13: A failed entity with blocked threads

Fault streams. While an entity is alive, its fault stream is alive. This is a consequence of the basic rule we mentioned above: the entity keeps its mediator alive, which itself keeps the entity's fault stream alive. Now consider the situation depicted in Figure 8.12 on the previous page. The site has no reference to the entity, but it still monitors it, with a thread suspended on the entity's fault stream.

The garbage collection process must implement the policy given in Section 4.4 on page 54. First, the entity may be removed from memory, since it is not referred to. Second, if the entity is removed, its fault stream must be closed with its tail bound to `nil`. Binding the stream tail should wake up the monitoring threads suspended on it. In order to be effective, it requires both the stream tail and its suspensions to be alive. Therefore, the fault stream of an entity must be kept alive until it is closed. The implementation is very simple, we add the step:

1.2. The fault streams of entities are taken as roots for the local garbage collector.

Blocked threads. We are now interested in the threads that block on a failed entity. Technically those threads are suspended on the control variable used by the entity's mediator to resume from temporary failures. The situation is illustrated in Figure 8.13. Let us recall the principle stated in Section 4.4 on page 54. If the failure is temporary, those threads must be kept alive, otherwise they would not resume. If the failure is permanent, they are not kept alive by the entity itself. We can easily ensure the liveness of those threads by adding the following step:

1.3. The control variables used by the mediators of temporary failed entities

are taken as roots for the local garbage collector.

Distributed operations. Those operations require to handle two extra data structures: components representing suspended operations and terms being serialized (see Figure 8.8 on page 119). Both are freed explicitly by the DSS; meanwhile, they must maintain their associated virtual machine entities alive. In other words, both suspended operations and terms being serialized, can be considered as roots for the local garbage collector. So we add the step:

1.4. Entities referred to by suspended operations and serialized terms are taken as roots for the local garbage collector.

9 Evaluation

This chapter gives an evaluation of our work. Both the language definition and its implementation are evaluated.

9.1 Ease of programming

In Chapter 5 we have shown a few abstractions built with our distribution model. From these examples, we can already say that both the customization of a distributed application and their handling of partial failure are not difficult. The examples show that nontrivial abstractions are coded relatively easily, and without too much code.

9.2 Performance

In this section we compare the performances of Mozart/DSS and Mozart. We will see that their performances are similar, with Mozart/DSS being a bit slower than Mozart. But the new protocols made available by Mozart/DSS permit to take better advantage of the distribution of Oz than Mozart. These performance comparisons complete the experiments done by Erik Klintskog on an early version of Mozart/DSS in [KBBH03]. These experiments showed that the Distribution Subsystem (DSS) incurred around 12% overhead on the total time for a client to perform a given number of requests to a server, when compared to Mozart. The same experiment also showed that Mozart/DSS was about twice as slow as an equivalent C++ program, optimized for the experiment, and using raw sockets for communication.

Issues. The execution of the performance tests was considerably delayed by bugs we discovered in the implementation. Not all those bugs could be fixed.

```
proc {Server N Len}
   S ServerPort={NewPort S}
in
   {Offer ServerPort ...}   % make ServerPort available
   for I in 1..N   get(X) in S do
      L={List.number I+1 I+Len 1}   % list of Len integers
   in
      X=L
   end
end

proc {Client N}
   ServerPort={Take ...}   % connect to the server's port
in
   for I in 1..N do X in
      {Send ServerPort get(X)}
      {Wait X}
   end
end
```

Snippet 9.1: Mozart/DSS vs. Mozart: server and client

Some of them were found in code that we did not write ourselves, notably in the DSS, which made the debugging task quite difficult. So the experiments we show here are the ones that could be run without triggering the remaining bugs.

9.2.1 Mozart/DSS vs. Mozart

This section compares the performance of the distribution layers in both platforms Mozart and Mozart/DSS. We consider a simple client-server program, and compute the CPU time spent in the distribution layer. Note that we do *not* measure network delays in this case. The client sends `N=100000` requests of the form `get(X)` to the server, and the server replies by binding `X` to a list of `Len` elements. The number `N` of requests has been chosen large enough in order to trigger the garbage collector, so that all the components of the distribution layer are involved in the test. We made experiments for `Len=10` and `Len=1000` to compare between small and big messages.

Snippet 9.1 above gives the code for the server and client. The server offers a port for the client to communicate. The procedure `Offer` creates a ticket, and makes it available (in a file, or on a web site). The procedure `Take` retrieves the ticket, and connects to it. The client sends `N` requests, and waits for the reply, so that requests are sequential. For the sake of simplicity, we made the server wait for `N` requests, then exit.

		Mozart	Mozart/DSS	
`Len=10`	centralized	0.260	0.292	
	distributed	25.500	29.304	
	difference	25.240	29.012	(+16%)
`Len=1000`	centralized	6.354	6.378	
	distributed	53.080	80.910	
	difference	46.726	74.532	(+60%)

Table 9.1: Comparison of total CPU times between platforms (network delays not included in measurements)

Results. Table 9.1 gives the results of our experiments. The numbers are the total CPU times spent by client and server in different situations. Network delays are not taken into account. Times are measures in seconds, and averaged over 5 executions. The programs were run on a computer with an Intel Core Duo 2 GHz processor. The centralized case corresponds to the client and server being run on a single site. In the distributed case, they are created on distinct sites, and their CPU time are added. The difference between the two reflects the global overhead in time of the distribution for that program.

The obvious observation is that Mozart/DSS is slower than Mozart, but the ratio between the two is reasonable. The slower performance of Mozart/DSS can be partly explained by its higher degree of flexibility, which requires a few more indirections when performing an entity operation. Another observation is that the size of shared data also has an impact on the relative performance of both platforms: the marshaling process is a bit more involved in Mozart/DSS, because the existing Mozart marshaler is wrapped to comply to the generic API defined by the DSS.

9.2.2 Comparing protocols

We show how we can dramatically change the network behavior of a simple program by changing how a distributed entity is annotated. The program used in the experiment is a simplistic chat application: peers join a group, and broadcast messages within that group. Each peer provides a port on which other peers send messages. The group itself is handled by a small server, which provides a cell with the list of ports. Message broadcast is done by sending the message to all the ports in the list. A peer joins the group by connecting to the server, and adding its port to the list in the cell.

Note that the distributed cell is not fault-tolerant. The purpose of the program is only to show performance variations within a program, just by changing distribution parameters. In this case, we will compare two protocols for distributing the cell.

Snippet 9.2 on page 133 gives the code for the group server and the group peers. The server is instantiated with a particular protocol for the cell. A peer

protocol	*total time*
migratory	63.906
replicated	11.775
no distribution (all peers on one site)	10.754

Table 9.2: Comparison of total time to complete (network delays included)

is created with a nickname. It first subscribes to the group. It then waits for one message to arrive on its port, then sends 1000 messages, with a pause of 10 milliseconds between two messages, unsubscribes and exits. Waiting for a message permits to synchronize peers, so they all start sending messages at the same time. Typically the last connected peer starts the process by calling `{Broadcast start}`.

The code to pay attention to is the procedure `Broadcast`. Each call to `Broadcast` reads the contents of the cell `Group`, and the procedure is called many times. Therefore the efficiency of the peer is largely influenced by how fast it can read `Group`.

Results. Table 9.2 gives the results of our experiments. The experiment consists in n peers broadcasting messages, with one of them also playing the role of the group server. The value $n = 3$ was sufficient to emphasize differences between protocols. The last connected peer broadcasts a dummy message to synchronize all peers. We measured the total time for one of the peers to run completely, i.e., the elapsed time between its startup and termination. In this case, we *do* measure network delays.

The experiment was run on Monday 5th November 2007, between 16:00 and 16:30. The times are measured in seconds and averaged over 5 runs. The peers were located on three machines: `calc6.info.ucl.ac.be` (everlab cluster at UCL), `planet8.cs.huji.ac.il` (everlab cluster at the Hebrew University of Jerusalem, Israel), and my own laptop Apple MacBook Pro connected at the UCL network. The server was located together with the peer on the first machine. The times were measured on the peer on the second machine.

As one can see, the migratory protocol, which is the default for distributing cells, is not efficient for that application. On the contrary, the replicated protocol, where each site has a copy of the current state of the cell, is almost as efficient as if all peers were on the same site. The reason is that the cell is read more often than it is updated, and the replicated protocol requires few network communication in that case. The choice of protocol has a noticeable impact on the performance of that example.

```
proc {Server Protocol}
   Group={NewCell nil}
in
   {Annotate Group Protocol}     % annotate cell
   {Offer Group ...}             % make Group available
end

proc {Peer NickName}
   Group={Take ...}       % connect to the cell Group
   proc {Subscribe P}     % add P to the group
      T in T=Group:=P|T
   end
   proc {Unsubscribe P}   % remove P from the group
      L T in
      L=Group:=T
      T={List.subtract L P}
   end
   proc {Broadcast M}     % send M to all ports in the group
      for P in @Group do
         {Send P M}
      end
   end
   S P={NewPort S}        % this peer's port
in
   thread
      for X in S do {Show X} end     % show all messages
   end

   {Subscribe P}
   {Wait S}                          % wait for start signal
   for I in 1..1000 do
      {Broadcast NickName#I}
      {Delay 10}
   end
   {Unsubscribe P}
end
```

Snippet 9.2: Comparing protocols: server and peer

10 Conclusion

10.1 Achievements

This work extends former studies on network transparency in distributed programming languages, by showing that this approach to distribution is practical regarding both efficiency and failure handling. We have given a few guidelines on how to structure a distributed program, and how to reason about its network behavior. We have extended the language Oz with annotations, that make the programmer able to customize the distribution of a program by choosing between distribution protocols for entities. The resulting program is a valid centralized program, and all distributed executions of the program are valid in a centralized setting.

We have also redesigned failure handling in Oz, made it simpler and more modular. Failure handling is based exclusively on asynchronous failure detection. We have introduced the concept of a fault stream to monitor an entity, and showed that this concept is sufficient to implement complex failure handling algorithms. The design of the fault stream also provides an effective post-mortem finalization mechanism, which was missing in the language. We have introduced new language operations to make entities fail. Those operations give the programmer more control to handle partial failures, for example by propagating the failure to a group of related entities.

On the implementation side, we have completed Erik Klintskog's work by making all protocols of the Distribution Subsystem (DSS) handle partial failure. Finally we have reimplemented the distribution of the platform Mozart on top of the DSS. The new implementation, Mozart/DSS, implements our distribution model for the language. We have been able to test it, and validate its effectiveness. However, the implementation is a prototype, and still suffers from quite a few bugs.

10.2 Future directions

Decentralized applications. This work should be a solid foundation to make applications fully decentralized. Some existing work, using structured overlay networks, have proved to be useful in this domain. Boris Mejías and Donatien Grolaux have already started to reimplement the library P2PS, which implements a structured overlay network in Oz [MCV05]. The new distribution model seems to be promising for that implementation.

A better virtual machine. The implementation of the virtual machine of Mozart is far from being simple. It is written in the language C++, but does not make use of object polymorphism, for instance. It is therefore difficult to maintain and to extend. It lacks modularity.

This lack of modularity is visible in the Glue layer: every language operation must be *explicitly* mapped to a DSS operation. For instance, the Oz cell has two *write* operations: `Exchange` and `Assign`. For each one, we had to provide a mapping to a DSS *write* operation, together with specific callbacks to perform those operations remotely or resume them. A better option would be to define only one *write* operation on cells, and both `Exchange` and `Assign` could be expressed in terms of that operation. All *write* operations in the virtual machine could be given the same distribution support, and polymorphism would allow to use the entity's *write* operation as a callback directly.

The DSS roughly defines five entity operations: *read*, *write*, *send*, *bind*, and *update*. Every language operations in Oz can be expressed as an instance of one of these operations. A mapping of the five generic operations would provide distribution support for all language operations. Of course this support has to be carefully designed, in order to avoid bad performance of elementary language operations, such as the sum operation.

Protocols written in Oz. In the DSS library, all entity protocols are written in C++. The existing protocols have limitations, for instance they have no recovery mechanism in case of failure. An Oz programmer cannot define its own protocol for objects, unless it modifies the underlying library and recompiles the platform. The recommended strategy is to define an Oz abstraction that encapsulates the recovery mechanism, and has its own protocol defined with ports and variables. This reduces the overall usefulness of the underlying library, since few protocols are really crucial.

Together with my colleague Yves Jaradin, we have sketched the design of a virtual machine that can be extended in the language itself. The idea is to give the possibility for an entity to delegate an operation to another entity, for instance a port. A distributed cell could be implemented by coordinating a group of cells on different sites, such that they give the illusion of being a unique cell. For each cell in the group, basic operations are delegated to a local agent, which can send messages (Oz values) to the other sites. The global

identity of the cell can be provided by an Oz name. The virtual machine should support serialization, and the possibility to send values to other sites.

The design has several advantages. First, it provides a framework to experiment with complex entity protocols at the language level. The latter protocol could use a user-defined overlay network to implement the communication between the sites that coordinate for an entity. Second, it gives the possibility to *dynamically upgrade* a protocol. Indeed, the code of the protocol is defined in the language as a value (a procedure, a class, or a functor), which can be send to a network of sites. A new protocol for cells can implemented and deployed without recompiling the virtual machine platform.

Security. Oz is relatively close to the language E, as we can easily encapsulate values and restrict their access via lexical scope. We can define *capabilities* in Oz in a quite reliable way. However, the implementation is more permissive than the language, in particular when distribution is in the game. The author has the impression that, with a reasonable effort, the distribution layer of Mozart/DSS can be made more secure. Some work was already done by Zacharias El-Banna and Erik Klintskog to make the DSS more secure [EKB05].

Besides that, Mozart/DSS also provides some tools at the language level. One can force all mutable entities to be stationary by default, for instance. This prevents a third-party site from screwing up an entity by cheating with its protocol.

A

Summary of the model

This chapter summarizes the distribution model, and all the language extensions introduced in this thesis.

A.1 Program structure

A program is distributed by letting several *sites* share language entities. The latter are stateless or stateful *data*. The basic operations on those entities behave as in the centralized case, modulo some network latency.

The program is deployed over the network by connecting several centralized programs with shared entities. A reference to an entity `X` can be converted from and to an atom `T` with the following functions. The atom is sent between sites by other means (e-mail, web site, etc.)

`{Connection.offer E}` returns an atom `T`.

`{Connection.take T}` returns the entity `E` from which the atom `T` was created with the former function.

Annotations. Entities can be *annotated* with protocol descriptions. The annotation states which kind of protocol should be used to distributed the entity. An entity's annotation cannot be modified.

`{Annotate E A}` annotates entity `E` with value `A`. Raises an exception if the value `A` is not valid for `E`.

Possible values for `A` are:

- access architecture: `access(stationary)`, `access(migratory)`
- entity protocols: `stationary`, `migratory`, `pilgrim`, `replicated`, `variable`, `reply`, `immediate`, `eager`, `lazy`

- garbage collection: `persistent`, `refcount`, `lease`

A.2 Failure handling

Entity failures. An entity fails by crashing: it stops being functional (forever). It can also fail locally on a given site: the entity is crashed on that site, but it can be correct on other sites. Operations on failed entities simply block.

Entity fault states and fault stream. The language provides failure detectors for entities. Those detectors define the following *local fault states* for the entity:

- `ok`: no failure detected
- `tempFail`: entity suspected of having failed, may go back to `ok`. Language operations block on the entity, and resume if the fault state goes back to `ok`.
- `localFail`: entity has failed locally
- `permFail`: entity has failed globally

Those states are notified in the *fault stream* of the entity. This stream reflects the sequence of fault states of the entity. It is accessed with the following function.

`{GetFaultStream E}` returns the fault stream of entity `E`.

The fault streams of two variables are merged when those variables are unified. The tail of the fault stream is bound to `nil` once the entity is no longer in memory. This can be used to program *post-mortem finalization*.

Making entities fail. Two operations are provided:

`{Kill E}` makes `E` fail globally. The operation is asynchronous, and is not guaranteed to succeed.

`{Break E}` makes `E` fail locally. The fault state of `E` becomes `localFail` immediately (unless it was already failed).

Bibliography

[AM03] Mostafa Al-Metwally. *Design and Implementation of a Fault-Tolerant Transactional Object Store*. PhD thesis, Al-Azhar University, Cairo, Egypt, December 2003.

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[AWWV96] J. Armstrong, M. Williams, C. Wikström, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, N.J., 1996.

[Bra05] Per Brand. *The Design Philosophy of Distributed Programming Systems: the Mozart Experience*. PhD thesis, Royal Institute of Technology (KTH), Kista, Sweden, 2005.

[BVCK00] Per Brand, Peter Van Roy, Raphaël Collet, and Erik Klintskog. Path redundancy in a mobile-state protocol as a primitive for language-based fault tolerance. Research Report RR2000-01, Université catholique de Louvain, Département INGI, 2000.

[Car95] Luca Cardelli. A language with distributed scope. In *Principles of Programming Languages (POPL)*, pages 286–297, January 1995.

[Col04] Raphaël Collet. Laziness and declarative concurrency. 2nd Workshop on Object-Oriented Language Engineering for the Post-Java Era: Back to Dynamicity PostJava'04, 2004.

[CV06] Raphaël Collet and Peter Van Roy. Failure handling in a network-transparent distributed programming language. In C. Dony et al., editor, *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 121–140. Springer-Verlag, 2006.

[EKB05] Zacharias El Banna, Erik Klintskog, and Per Brand. Making the distribution subsystem secure. Technical Report T2004:14, Swedish Institute of Computer Science, Kista, Sweden, January 2005.

[GGV04] Donatien Grolaux, Kevin Glynn, and Peter Van Roy. A fault tolerant abstraction for transparent distributed programming. In *Second International Mozart/Oz Conference (MOZ 2004)*, Charleroi, Belgium, October 2004. Springer-Verlag LNCS volume 3389.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. Available at `http://java.sun.com` (June 2007).

[GLT97] H. Guyennet, J.-C. Lapayre, and M. Tréhel. Distributed shared memory layer for cooperative work applications. In *22nd Annual Conference on Computer Networks, LCN'97*, pages 72–78, Minneapolis, USA, November 1997. IEEE Computer Society and TC Computer Communications.

[GR06] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, 2006.

[HVB+99] Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3):569–626, May 1999.

[HVBS98] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.

[HVS97] Seif Haridi, Peter Van Roy, and Gert Smolka. An overview of the design of Distributed Oz. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97)*, pages 176–187, Maui, Hawaii, USA, July 1997. ACM Press.

[ISO98] ISO/IEC. Open distributed processing - reference model: Overview, 1998.

[Jul88] Eric Jul. *Object Mobility in a Distributed Object-Oriented System*. PhD thesis, Univ. of Washington, Seattle, Wash., 1988.

[KBBH03] Erik Klintskog, Zacharias El Banna, Per Brand, and Seif Haridi. The design and evaluation of a middleware library for distribution of language entities. In *Asian Computing Science Conference*, pages 243–259, 2003.

[Kli05] Erik Klintskog. *Generic Distribution Support for Programming Systems*. PhD thesis, Royal Institute of Technology (KTH), Kista, Sweden, 2005.

[KNBH01] Erik Klintskog, Anna Neiderud, Per Brand, and Seif Haridi. Fractional weighted reference counting. In Rizos Sakellariou, John Keane, John R. Gurd, and Len Freeman, editors, *Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, Proceedings*, volume 2150 of *Lecture Notes in Computer Science*, pages 486–490. Springer, 2001.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 1979.

[MCGV05] Valentin Mesaros, Raphaël Collet, Kevin Glynn, and Peter Van Roy. A transactional system for structured overlay networks. Research Report RR2005-01, Université catholique de Louvain, Département INGI, 2005.

[MCV05] Valentin Mesaros, Bruno Carton, and Peter Van Roy. P2PS: Peer-to-peer development platform for Mozart. In Peter Van Roy, editor, *Multiparadigm Programming in Mozart/Oz: Extended Proceedings of the Second International Conference MOZ 2004*, volume 3389 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2005.

[Mil06] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[Moz99] The Mozart programming system (Oz 3), January 1999. Available at `http://www.mozart-oz.org` (June 2007).

[Sah04] Per Sahlin. Efficient distribution of immutable data structures in the distributed subsystem middleware library. Master's thesis, Royal Institute of Technology (KTH), Kista, Sweden, 2004.

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[SCV03] Alfred Spiessens, Raphaël Collet, and Peter Van Roy. Declarative laziness in a concurrent constraint language. 2nd International Workshop on Multiparadigm Constraint Programming Languages MultiCPL'03, 2003.

[Smo95] Gert Smolka. The Oz programming model. In *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[Sun97] Sun Microsystems. *The Remote Method Invocation Specification*, 1997. Available at `http://java.sun.com` (June 2007).

[Van06] Peter Van Roy. Convergence in language design: A case of lightning striking four times in the same place. In *Functional and Logic Programming, 8th International Symposium (FLOPS)*, volume 3945 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 2006.

[VBHC99] Peter Van Roy, Per Brand, Seif Haridi, and Raphaël Collet. A lightweight reliable object migration protocol. *Lecture Notes in Computer Science*, 1686:32–46, 1999.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, 2004.

[VHB+97] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in Distributed Oz. *ACM TOPLAS*, 19(5):804–851, September 1997.

[VHB99] Peter Van Roy, Seif Haridi, and Per Brand. *Distributed Programming in Mozart – A Tutorial Introduction*, 1999. In Mozart documentation, available at `http://www.mozart-oz.org` (June 2007).

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Mountain View, CA, November 1994.