An Overview of the Design of Distributed Oz

Seif Haridi
Swedish Institute of Computer Science (SICS), Sweden
seif@sics.se

Peter Van Roy
Université catholique de Louvain (UCL), Belgium
pvr@info.ucl.ac.be

Gert Smolka
Universität des Saarlandes, Saarbrücken, Germany
smolka@ps.uni-sb.de

Abstract

We present a design for a distributed programming system, Distributed Oz, that abstracts away the network. This means that all network operations are invoked implicitly by the system as an incidental result of using particular language constructs. However, since network operations are expensive, the programmer must retain control over network communication patterns. This control is provided through the language constructs. While retaining their centralized semantics, they are extended with a distributed semantics. Distributed Oz is an extension of Oz, a concurrent state-aware language with first-class procedures. Distributed Oz extends Oz with just two concepts: mobility control and asynchronous ordered communication. Mobility control provides for truly mobile objects in a simple and clean way. Asynchronous ordered communication allows to conveniently build servers. These two concepts give the programmer a control over network communications that is both simple and predictable. We give scenarios to show how common distributed programming tasks can be implemented efficiently. There are two reasons for the simplicity of Distributed Oz. First, Oz has a simple formal semantics. Second, the distributed extension is built using network protocols that are natural extensions to the centralized language operations. We discuss the operational semantics of Oz and Distributed Oz and the architecture of the distributed implementation. We give an overview of the basic network protocols for communication and mobility.

Keywords. Language-based Distribution, Mobile Objects, Network Transparency, Semantics, Implementation, Concurrent Constraints

1 Introduction

The number of computers in the world and their interconnectedness both continue to increase at an exponential rate [17]. This leads one to predict that eventually most applications will run across a network, i.e., will be distributed. To manage the increasing complexity of application development, we propose that a long-term solution to distributed programming must reduce its programming complexity to be comparable to centralized (i.e., non-distributed) programming. We propose to achieve this by designing and implementing a language, Distributed Oz, that *abstracts away the network*. All network operations are then invoked implicitly by the system. However, for efficiency the programmer should retain control over network communication patterns. In Distributed Oz there are no explicit operations (such as message sending or receiving) to transfer data. All network transfers are done implicitly through the same language constructs of centralized programming. In order for this to be practical, these language constructs must be extended to a distributed environment such that network communication patterns are clearly perceived by the programmer. For example, the language must give clear meanings to replicated data and mobile objects.

1.1 Extending the basic language operations

All systems that we know of except Obliq [3] and Emerald [13] do distributed execution by adding a distribution layer on top of an existing language [18, 26, 27, 30]. This has the disadvantage that distribution is not a seamless extension to the language, and therefore distributed extensions to language operations (such as mobile objects or replicated data) must be handled by explicit programmer effort. A better technique is to look carefully at the basic operations of the language and to conservatively extend them to a distributed setting. For example, the Remote Procedure Call (RPC) [26] is designed to mimic centralized procedure calling and is therefore a precursor to the design given in this paper. This second approach has two consequences. First, in order to carry it out successfully, the language must have a well-defined operational semantics that clearly identifies the basic operations and separates them from each other.

Second, to do the distributed extensions right one must design a network protocol for each basic operation of the language. Obliq has taken a first step towards this goal. Obliq distinguishes between *values* and *locations*. Moving values causes them to be copied (replicated) between sites. Moving locations causes network references to them to be created. Distributed Oz goes further towards this goal by taking this approach for the complete language. Distributed Oz distinguishes between variables, records, procedures, cells and ports. Cells are used to implement concurrent object state. Ports are used to implement many-to-one and many-to-many channels. Each of these entities has a network protocol that is used whenever a basic operation is performed on it. The network protocols are designed to preserve language semantics while providing a simple model for the communication patterns.

1.2 Oz as the foundation

The Oz language appears to the programmer as a concurrent object-oriented language that is state-aware and has dataflow synchronization [8].[1] In a broader context, Oz is a successful attempt to integrate several programming paradigms (including object-oriented, functional, and constraint logic) into a simple model [22]. This model allows to understand various forms of computation as facets of a single phenomenon. This increases the range of problems that can easily be cast into the language [23]. Oz has a fully defined formal semantics [24] as well as an efficient and robust implementation [19]. Oz has been used in many research projects [1, 5, 6, 7, 12, 11, 21, 29].

We take Oz as the foundation of our design because it provides four key concepts in a clear and formal way. First, Oz is a *concurrent* language, which is necessary since distributed systems are inherently parallel. Second, Oz has *dataflow synchronization*. Threads block when the data they require is not yet available. This blocking is invisible from the point of view of the thread. Third, Oz has *first-class procedures* with *lexical scoping*. This means that mobile procedures and objects will behave correctly no matter where they are executed. One might say that they take their environments with them. Fourth, Oz is *state-aware*: it makes a distinction between stateless and stateful data. Stateless data (which does not change) is replicated between sites. Stateful data (which can change) is the basis of an object system that models stationary servers and mobile agents as facets of the same concept (they are both objects).

1.3 Distributed Oz is almost Oz

To the programmer, a Distributed Oz program is almost identical to an Oz program. The latter is converted to the former with only minor modifications to the source text. For example, the graphic editor of Section 2 is made distributed by specifying the mobility behavior of the objects used to represent the picture state. In a first approximation, Distributed Oz extends Oz with just *two* notions specific to distribution: *mobility control* and *asynchronous ordered communication*. These concepts are defined in Section 4.2.

1.4 Organization of the paper

This paper is organized into five parts. Section 2 motivates the design by means of an example application written in Distributed Oz. Section 3 presents a set of high-level requirements that we assume the design must satisfy. The section then determines for each requirement what mechanisms are needed to achieve it. Section 4 defines the semantics of a language, Distributed Oz, whose language constructs provide these mechanisms. The centralized semantics of Distributed Oz are given in terms of a model called OPM (Oz

[1]The constraint programming part of Oz is outside the scope of this paper.

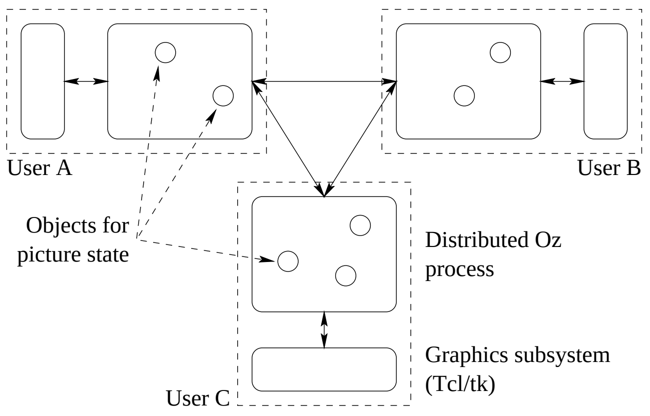

Figure 1: A shared graphic editor

Programming Model). Section 5 gives examples to illustrate programming in Distributed Oz. Finally, Section 6 summarizes the graph model of distributed OPM execution, the mobility protocols for stateful entities (cells and ports), which are defined in terms of this model, and the model's implementation architecture.

2 A shared graphic editor

Writing an efficient distributed application can be much simplified by using transparency and mobility. We have substantiated this claim by designing and implementing a prototype shared graphic editor, an application which is useful in a collaborative work environment. The editor is seen by an arbitrary number of users. We wish the editor to behave like a shared virtual environment. This implies the following set of requirements. We require that all users can make updates to the drawing at any time, that each user sees his own updates without any noticeable delays, and that the updates must be visible to all users in real time. Furthermore, we require that the same graphical entity can be updated by multiple users. This is useful in a collaborative CAD environment when editing complex graphic designs. Finally, we require that all updates are sequentially consistent, i.e., each user has exactly the same view of the drawing at all times. The last two requirements are what makes the application interesting. Using multicast to update each user's visual representation, as is done for example in the wb whiteboard [14], does not satisfy the last two requirements.

Figure 1 outlines the architecture of our prototype. The drawing state is represented as a set of objects. These objects denote graphical entities such as geometric shapes and freehand drawing pads. When a user updates the drawing, either a new object is created or a message is sent to modify the state of an existing object. The object then posts the update to a distributed agenda. The agenda sends the update to all users so they can update their displays. The users see a shared stream, which guarantees sequential consistency.

New users can connect themselves to the editor at any time. The mechanism is extremely simple: the implementation provides primitives for saving and loading a language entity in a file named by a URL. We use URLs because of convenience. They are Ascii strings that are part of a global naming service that is available on any site that has an HTTP server. The graphic editor saves a reference to the object responsible for managing new users. A new user loads a reference to this object. The two computations then share a reference to the same entity. This opens a connection between two sites in the two computations. From that point onward, the computation space is shared. When there are no more references between two sites in a computation, then the connection between them is closed by the garbage collector. Computations can therefore connect and disconnect at will. The issue of how to manage the shared names represented by the URLs leads us into the area of distributed multi-agent computations. This is beyond the scope of the paper.

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

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

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

In general, mobile objects are useful both for fine-grain mobility (caching of object state) as well as coarse-grain mobility (explicit transfer of groups of objects). The key ability that the system must provide is transparent control of mobility. Section 5.1 shows how this is done in Distributed Oz.

3 Requirements and mechanisms

As a first step in the design process, we determine what the requirements are for our distributed system. We assume the system must satisfy at least four requirements: network transparency, network awareness, latency tolerance, and language security. Table 1 gives these requirements and the mechanisms that we provide to achieve them. For brevity, we omit in this paper the discussion of other requirements such as resource localization, multicast support, implementation security and fault tolerance.

Requirements	Mechanisms
Network transparency	Shared computation space Concurrency
Network awareness	State awareness Mobility control
Latency tolerance	Concurrency Logic variable
Language security	Lexical scoping First-class procedures

Table 1: System requirements and their mechanisms

3.1 Network transparency

Network transparency[2] means that computations behave in the same way independent of the distribution structure. The language satisfies the centralized semantics defined in Section 4.1. This requires an *shared computation space*, which provides the illusion of a single network-wide memory for all data values (including objects and procedures). The distinction between local references (on the same site) and remote references (to another site) is invisible to the programmer. Furthermore, a program running on many sites (i.e., in parallel) must behave in the same way when running on a single site. Therefore we must provide *concurrency*, that is, to allow multiple computational activities that coexist and evolve independently.

3.2 Network awareness

Network awareness means that network communication patterns are simply and predictably derivable from the primitive language operations. The resulting model gives the programmer explicit control over network communication patterns. The language satisfies the distributed semantics defined in Section 4.2. To achieve network awareness, the basic insight is that stateful data (e.g., objects) must reside on one site[3] and that stateless data (e.g., records) can be replicated. The system must therefore distinguish between these two kinds of data, that is, it is *state-aware*. For stateful data, we need control over its location. We introduce the concept of *mobility control*, which is the ability for entities to migrate between sites or to remain stationary at one site.

With the concepts of state awareness and mobility control, the programmer has control over network communication patterns. Mobile entities migrate to each remote site invoking them. For predictability, mobile entities must not leave a trail. This is implemented using a mobility protocol (see Section 6.2). Stationary entities require a network transaction on each remote invocation. A replicable entity can either be copied implicitly whenever a network reference would be created (*eager* replication) or the network reference is created and the entity is copied on request (*lazy* replication). Mobile and lazy replicable entities require a network transaction only on the first access.

[2]The terms "network transparency" and "network awareness" were first introduced by Cardelli [3].

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

3.3 Latency tolerance

Latency tolerance means that the efficiency of computations is as little affected as possible by the latency of network operations. In this paper, we assume that this means that computations are concurrent and stall only on data dependency (not on send or receive). *Concurrency* provides latency tolerance between threads: while one thread waits for the network, the other threads continue. To stall only on data dependency, we provide *logic variables*. The logic variable describes an entity that is possibly not yet known.[4] Communicating and synchronizing between two threads can be done with a shared logic variable. The first thread puts a constraint on the variable and the second thread has a conditional on the variable. Logic variables provide latency tolerance within threads by decoupling the operations of calculating and using the value (which define a data dependency) from the operations of sending and receiving the value.

3.4 Language security

Language security means that given secure communications, the language guarantees integrity of computations and data. It is important to distinguish between *language* security and *implementation* security. The latter means that integrity of computations is protected against adversaries that have access to the system's implementation. We provide language security by giving the programmer the means to restrict access to data, through lexical scoping and first-class procedures. *Lexical scoping* means that one can only access data to which one has an explicit reference; and that the initial references are determined by the program's static structure. In particular, one cannot address data indirectly through address calculation or dynamic binding. *First-class procedures* means that procedures are a kind of data value and can be passed around. Procedures can encapsulate references and restrict operations on them. Passing procedures around thus transfers access rights, which gives the effect of capabilities.

4 Semantics of Distributed Oz

The semantics of Distributed Oz are defined by reduction to a simple kernel language, called the Oz Programming Model (OPM). OPM is carefully designed to factor out the different concepts of Oz and is therefore quite a simple model.[5] OPM in fact gives the semantics of Oz, which is a centralized system. We therefore give the semantics of Distributed Oz in two parts. First, we define the *centralized semantics*, which coincides with the semantics of Oz. Then, we define the *distributed semantics*, which defines what network operations are invoked by the operations of OPM when the computation is partitioned on multiple sites.

4.1 Centralized semantics

The centralized semantics of Distributed Oz defines the operational behavior when sites are disregarded. This semantics is identical to the operational semantics of Oz, which is given by the Oz Programming Model (OPM). This section presents OPM and discuss its operational semantics. The full formal treatment is not much more complex than this discussion. We refer the reader to [24] for a formal definition of OPM.[6]

OPM is concurrent, provides synchronization through logic variables, provides higher-order procedures, and clearly distinguishes between stateless and stateful computation. The execution is defined as the reduction of *expressions* E. The data elements are modeled with *constraints* C. A constraint is a relation between data items that partially determines each data item's value. Representing data by constraints is important for latency tolerance. Furthermore, such a representation can be made very efficient. It allows sending and receiving a data item before its value is completely known. We give a BNF-style definition of the expression and constraint syntax together with a discussion of their semantics.

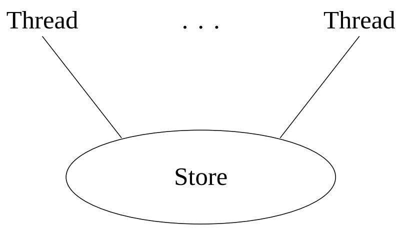

Figure 2: The basic execution model

4.1.1 Concurrent constraints

The basic model is similar to Saraswat's concurrent constraint model [20], providing for concurrent control and synchronization via logic variables.

$$\begin{array}{lcl} E & ::= & E;E \mid \textbf{thread}\ E\ \textbf{end} \mid \textbf{local}\ x\ \textbf{in}\ E\ \textbf{end} \mid \\ & & C \mid \textbf{if}\ C\ \textbf{then}\ E\ \textbf{else}\ E\ \textbf{end} \\ C & ::= & x = y \mid x = f(a_1 : y_1, ..., a_n : y_n) \mid C \wedge C \end{array}$$

As depicted in Figure 2, computation takes place in a *computation space* hosting a number of *threads* connected to a single shared *store*, each reducing an expression E. The store is partitioned into compartments for constraints, procedures, and cells. As computation proceeds, the store accumulates constraints on variables in the constraint store.

Computation proceeds by *reduction of expressions* that interact with the store and may create new threads. Once an expression becomes reducible, it stays reducible. Reduction is fair between threads. The basic control primitive for thread reduction is sequential composition $(E_1; E_2)$. A thread can thus be decomposed into a sequence of primitive *tasks*. Reduction of $(E_1; E_2)$ yields $(E'_1; E_2)$, where E_1 reduces to E'_1. E_2 is reduced only when E'_1 disappears. Therefore the control flow of a thread is strictly sequential. If an expression E representing a thread is not reducible then the thread *blocks*. Thread creation is expressed by **thread** E **end**. New variables (i.e., new references in the constraint store) are introduced by the declaration (**local** x **in** E **end**). The variable x is only visible inside E (lexical scoping). There are two primitives to manipulate data. First, putting a constraint in the store (C) is used to

[4]Like a box that may be empty or full. Boxes can be passed around without looking inside.

[5]In this paper we consider OPM with sequential composition.

[6]The constraint programming part of Oz (with search spaces and disjunctions) is outside the scope of this paper.

communicate information. The constraints represent common data types (records, numbers, strings, etc.) which we group together under the collective name *record*. The new information is only put in the store if it is not inconsistent with what is already there. In the contrary case, the store is unchanged and an exception is raised in the thread containing the constraint. Second, waiting for a constraint to appear (**if** C **then** E_1 **else** E_2 **end**) is used to synchronize.

4.1.2 First-class procedures

Procedures are the basic primitives for abstracting computation. Procedures are triples of the form $\xi{:}y_1...y_n/E$ consisting of a *unique name* ξ, *formal arguments* $y_1...y_n$ and a body E. Procedures may be created and applied as follows:

E ::= ... | **proc** {x $y_1...y_n$ } E **end** | {x $y_1...y_n$ }

Procedures are created by reducing a task of the form **proc** {x $y_1...y_n$ } E **end**. This binds the variable x to a freshly created name ξ and enters the procedure $\xi{:}y_1...y_n/E$ into the *procedure store*. There is one basic operation, namely procedure application {x $z_1...z_n$ }. This waits until the procedure store contains a procedure $\xi{:}y_1...y_n/E$ such that the constraint store contains $x = \xi$. Then, it can reduce to the task $E[z_1...z_n/y_1...y_n]$, which is obtained from the procedure body E by replacing the formal arguments $y_1...y_n$ by the actual ones $z_1...z_n$.

4.1.3 State

The model supports state by means of *cells*. A cell is a pair $\xi{:}x$ representing the mutable binding of a name ξ to a variable x. Cells may be created and updated as follows:

E ::= ... | `{NewCell` y x `}` | `{Exchange` x y z `}`

Cells are created by reducing the task `{NewCell` y x `}`. Similar to procedure definition, reduction of this task picks a fresh name ξ, binds x to ξ and enters the pair $\xi{:}y$ into the third compartment of the store, the *cell store*. The pair $\xi{:}y$ is called the cell's *content-edge*. There is one operation on cells, namely `{Exchange` x y z `}`. This waits until the cell store contains a cell $\xi{:}u$ and the constraint store contains $x = \xi$. In that case, there is an atomic transaction consisting of two parts: the cell binding is updated to $\xi{:}z$ and the exchange reduces to the expression $y = u$. The latter can then reduce in its turn. The effect is that y will be given access to the cell's old binding, and z is the cell's new binding. Using cells allows to model data that change arbitrarily over time (stateful data). In contrast, the constraint and procedure stores only accumulate data that are consistent with previous data (stateless data).

4.2 Distributed semantics

The distributed semantics are motivated by the desire to provide network awareness. The distributed semantics are defined as the network operations done by the primitives of OPM when the execution is done in a distributed setting, that is, the computation is partitioned on multiple *sites*. If the concept of site is disregarded, then the behavior coincides with the centralized semantics.

4.2.1 Mobility control

The basic concept that is added to the language is *mobility control*, as defined in Section 3. This property affects stateful entities, namely cells and ports. Ports are defined below; for now consider them to be similar to cells in that they contain a state and can update this state. Each entity has a set of basic operations, including a basic state-updating operation. This operation can cause the state to move to the invoking site, or to remain on the same site. In the first case, the entity is called *mobile*. In the second case, it is called *stationary*. In the current model, cells are *mobile* and ports are *stationary*.

A cell has one state-updating operation, exchange, which causes the state to move atomically to the site invoking it. A port has two operations, send and localize. Doing a send does not change the site of the port's state. Doing a localize causes the port's state to move atomically to the site invoking it. Section 5 shows how to use cells and ports to define objects and servers with arbitrary migratory behavior. Section 6.2 presents the mobility protocol for cells and ports.

In addition, we define records, procedures, and variables to be *replicable*. Since they are stateless, records and procedures can be copied to a site. In the current model, records and their arguments are copied eagerly i.e., a network reference to a record cannot exist. Procedures are copied only upon request. Both a procedure's code and its closure have a network address. Therefore a site has at most one copy of each code block or closure. Binding a variable causes it to be eliminated eagerly, that is, the fact of its being bound is transmitted eagerly to all its representatives on different sites. If it is bound to a record, then the record is copied eagerly to all the variable's sites.

Network awareness for OPM primitives is derived as follows from the above classification. A cell migrates to each site invoking it (through an `Exchange`). A port requires a network transaction on each remote send. Procedures are replicated lazily and records are replicated eagerly. Cells and procedures require a network transaction only on the first access.

```
proc {NewPort S Port}
   local C in
      {NewCell S C}
      proc {Port Message}
         local Old New in
            % Create next element of stream
            {Exchange C Old New}
            thread Old=Message|New end
         end
      end
   end
end

proc {Send Port X}
   {Port X}
end
```

Figure 3: Defining a port in terms of OPM

4.2.2 Asynchronous ordered communication

The basic primitive for asynchronous ordered communication is called the *port*. A port is the pair of an identifier `Port` and a stream (a list with unbound tail). Applying `{Send Port X}` appends `X` to the stream. We call this *sending* `X` to the port. Ports allow many-to-one and many-to-many communication. On the sending side, any thread with a reference to `Port` can send `X` to the port. Concurrent updates to the stream are serialized. On the receiving side, since the stream is stateless, any number of threads can wait for and read its elements concurrently. Ports can be created and updated as follows:

$$E \ ::= \ \ldots \mid \texttt{\{NewPort } s\ p\ \texttt{\}} \mid \texttt{\{Send } p\ x\ \texttt{\}}$$

A port can be defined in OPM as a procedure that references a cell. Figure 3 shows how to define `NewPort` and `Send`.

A port is created by reducing the task `{NewPort S Port}`. This creates `Port` and ties it to the stream `S`. A message `X` is sent to the port by `{Send Port X}`. This appends `X` to the stream and creates a new unbound end of the stream. One can build a *server* by creating a thread that waits until new information appears on the sequence and then takes action depending on this information.

We define the distributed semantics of a port's `Send` operation to be *asynchronous* and *ordered*. That is, `Send` operations from the same thread complete immediately and the messages appear on the stream in the same order that they were sent. We provide ports as a primitive since they support common programming techniques and the asynchronous ordering can exploit popular network protocols.

Distributed Oz provides three ways for threads to communicate with each other:

- In *synchronous communication* the sender waits for a response from the receiver before continuing. This is the most deterministic yet least efficient method if the threads are on different sites. For example, the sender invokes `{Send Server query(Q A)} {Wait A}` where `Q` is a query and `A` is the answer. Through `{Wait A}`, the sender waits until the receiver has instantiated the answer.
- In *asynchronous unordered communication* the sender does not wait, but the messages do not necessarily arrive at the receiver in the order sent. This is the least deterministic yet most efficient method if the threads are on different sites. For example, the sender invokes **thread** `{Send Server query(Q A)}` **end**.
- In *asynchronous ordered communication* the sender does not wait, yet the messages arrive at the receiver in the order sent. It is implemented by communicating through a port. For example, the sender can invoke `{Send P query(Q1 A1)} {Send P query(Q2 A2)}`. Query `Q1` will arrive at the receiver before query `Q2`, yet query `Q2` can be sent before an answer is received for `Q1`. A practical example would be a database server that receives queries from different threads and performs transactions. From the viewpoint of each thread, the transactions should be performed in the order they were sent.

5 Programming Idioms

5.1 Basic tools for the programmer

The primitives defined in Section 4 are part of the OPM semantics. Using these primitives, we can provide many useful derived concepts to the programmer. We give four typical examples here.

5.1.1 Concurrent objects

Cells plus first-class procedures are sufficient to define a powerful *concurrent object system* within OPM [10]. Semantically, objects are procedures that reference a cell whose content designates the current object state. Mutual exclusion of method bodies is supported through explicit thread-reentrant locking. A thread-reentrant lock allows the same thread to reenter the lock, i.e., to enter a dynamically-nested critical region guarded by the same lock. The implementation optimizes object invocations to make them as efficient as procedure calls.

5.1.2 Freely mobile concurrent objects

An object is called *freely mobile* if it is implemented by a mobile cell. When a site sends a message to the object, then the object will be moved to the sending site and the message will be handled there. The mobile cell semantics, implemented by the mobile cell protocol (see Section 6), guarantees a correct serialization of the object state. Freely mobile objects are optimized to reduce latency as much as possible: they are effectively cached at the sites that use them. Any further messages will be handled locally.

```
proc {MakeServer ObjQ ServQ}
   Str
    proc {ServeLoop S}
       if X Sx in S=X|Sx then
          {ObjQ X}
          {ServeLoop Sx}
       else
          {ObjQ close}
       end
    end
in
   {NewPort Str ServQ}
    thread {ServeLoop Str} end
end
```

Figure 4: A stationary server

5.1.3 Stationary servers

We define a stationary server as an object that is fixed to one site and to which messages can be sent asynchronously and ordered. Such a server can be defined as a port whose stream is read by a new thread that passes the messages to the object. For example, Figure 4 shows how an object `ObjQ` can be converted to a server called `ServQ`. A port is used to "protect" the object from being moved and from being executed by more than one thread at a time. Remotely invoking a stationary server is similar to a remote procedure call (RPC). It is more general because it may pass incomplete messages (logic variables) which can be filled in at some later time.

```
proc {MakeServer ObjQ ServQ Move}
   Str Prt Key
   proc {ServeLoop S}
      if Stopped Rest Sx in
         S=Key(Stopped Rest)|Sx then
         Rest=Sx
         Stopped= unit
      elseif X Sx in S=X|Sx then
         {ObjQ X}
         {ServeLoop Sx}
      else
         {ObjQ close}
      end
   end
in
   {NewName Key}
   {NewPort Str Prt}
   proc {Move}
   Stopped Rest in
      {Send Prt Key(Stopped Rest)}
      {Wait Stopped}
      {Localize Prt}
      thread {ServeLoop Rest} end
   end
   proc {ServQ M} {Send Prt M} end
   thread {ServeLoop Str} end
end
```

Figure 5: A mostly-stationary server

5.1.4 Mostly-stationary servers

A mostly-stationary server behaves in the same way as a server, except that an operation `Move` is defined that moves the server to the site invoking the move. The move is done atomically. After the move, it is eventually true that messages sent to the server will go directly to the new location in a single network hop. From this point onwards, no forwarding is done. A mostly-stationary server is similar to but more powerful than the "mobile agent" concept that has recently become popular (e.g., with Telescript [16]). Moving the server enables it to compute with resources of its new site. Like all entities in Oz, the server can access only the references that it has been explicitly given, which gives language security.

Figure 5 shows how an object `ObjQ` can be converted to a server called `ServQ` and a move operation `Move`. The server loop can be stopped by sending the stop message `Key(Stopped Rest)`, where `Key` is an Oz name used to identify the stop message and `Stopped` and `Rest` are outputs. Since `Key` is unforgeable and known only inside `MakeServer`, the server loop can be stopped only by `Move`. The port `Prt` must be hidden inside a procedure, otherwise it can be localized by any client. When the loop is stopped, `Rest` is bound to the unprocessed remainder of the message stream. The new server loop takes `Rest` as its input.

5.2 Scenarios

5.2.1 A distributed virtual reality system

The DIVE system (Distributed Virtual Environment) [7] provides views to a common virtual world from many sites. A basic operation of the environment is to make events in the virtual world visible from all the sites. An efficient way to do this is with a port. The messages written to the port contain updates to the virtual world. Each site reads the updates on the port's stream and uses them to update its visual representation of the virtual world. The port serializes the writers. Since the stream is stateless, an arbitrary number of readers can access it concurrently and they do not interfere with the writers. This desirable property is a consequence of the clear separation between stateful and stateless data. That is, one can have a stateful reference to stateless data. The writer updates the stateful reference. Because stateless data cannot change, the reader accesses it unbothered by the writer.

5.2.2 A client/server with mobility

Assume that a client wants to access a database server. The server is flexible and lets the client provide its own search procedure. The server is modeled by a port and the client's request by a mobile object. The request object encapsulates the client's search procedure as well as having a reference to accumulate the results. The client keeps the same reference. The client sends a message containing the request object to the server. When the server invokes the search, it transparently causes the request object to move to the server site. The client's search procedure is then executed on the server site. Through its reference, the client has access to the results as soon as they become available.

5.2.3 A distributed MUD

The MUD (Multi-User Dimension) shell *Munchkins* [9] is a multi-user game written in Oz and using the existing Internet libraries to provide a limited distribution ability. The Munchkins implementation allows transparent distribution of records and transparent message sending to remote objects through proxy objects. Porting Munchkins to Distributed Oz will simplify the mapping of different parts of its simulated world to different sites. Objects within the MUD world can be modeled as mobile objects in Distributed Oz. An object that moves in the MUD world will migrate in the Distributed Oz system. This will much improve the scalability of the MUD since objects will only use resources in the room that they are in.

6 Implementation concepts

The Distributed Oz implementation is based on a distributed execution model for OPM (see Table 2). A major design goal was that distribution should not hinder the centralized implementation. This has two main consequences. First, the implementation should be a minimal extension of the centralized implementation of Oz. Second, the extension must not affect the performance of non-distributed programs.

We first explain the graph model of distributed OPM execution. Then we present the mobility protocol for cells and ports. Full details on the part of the graph model dealing

Distributed Oz
⇓
Oz Programming Model
(semantic definition of Distributed Oz)
⇓
Language graph
(graph model of OPM)
⇓
Distribution graph
(language graph + notion of site)
⇓
Network protocols
(distributed algorithms between communicating nodes)

Table 2: Modeling the implementation of Distributed Oz

with mobility can be found in [28]. Finally, we present an overview of the implementation architecture.

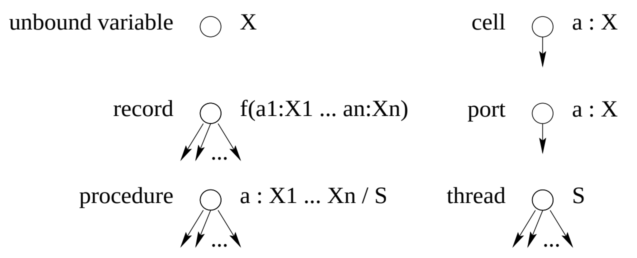

Figure 6: Nodes in the language graph

6.1 Graph models of Distributed Oz

We present a graph model of the distributed execution of OPM. An OPM computation space can be represented in terms of two graphs: a *language graph*, in which there is no notion of site, and a *distribution graph*, which makes explicit the notion of site. We then explain what mobility means in terms of the distribution graph. Finally, we summarize the failure model in terms of the distribution graph.

6.1.1 The language graph

We introduce distribution into the execution model of OPM in two steps. The first step, depicted in Figure 6, models the threads and store of OPM (see Section 4.1) as a graph called the *language graph*. The language graph of Oz was first introduced by Smolka *et al* [25]. The graph has six different node types: **thread**, **record**, **procedure**, **variable**, **cell**, and **port**. The five non-thread nodes are also called *data* nodes. A thread node points to its external references. A record node points to its arguments. A procedure node points to its external references (which lexical scoping has given it). A variable node points to the threads waiting on the variable. A cell node points to its value (which can be accessed through an exchange operation). A port node points to its value (the end of the stream).

Each reduction of an OPM expression corresponds to a graph transformation. These transformations can be effected by the nodes themselves by considering them to be concurrent objects that exchange messages along the edges of the graph. The nodes must contain sufficient state to enable them to perform the graph operations. For example, a procedure node contains the procedure definition and a task node contains the expression to be reduced.

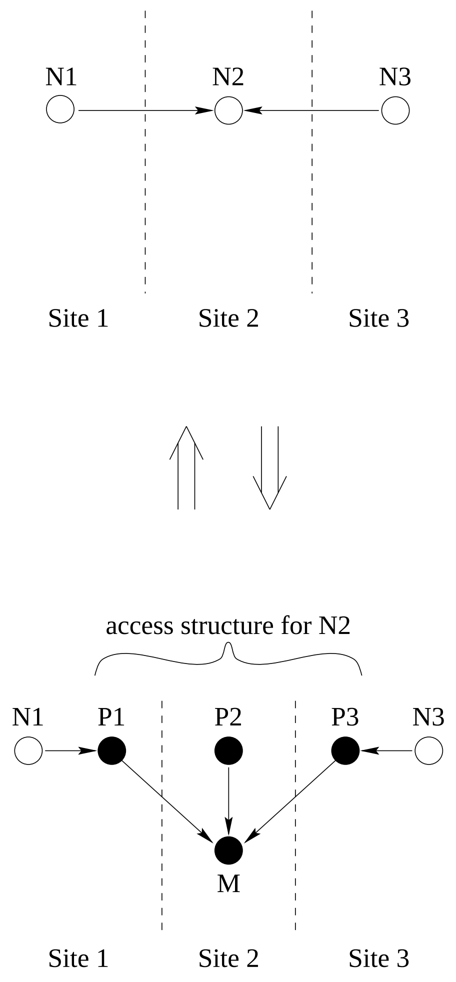

Figure 7: From language graph to distribution graph

6.1.2 The distribution graph

The second step is to extend the language graph with the notion of *site* (see Figure 7). Introduce a finite set of sites and annotate each node of the language graph with a site. If a node is referenced by a node on another site, then we map it to a *set* of nodes in the distribution graph. This set is called the *access structure* of the original node. The graph resulting after all nodes have been so transformed is called the *distribution graph*.

Access structures contain two new types of nodes, called *managers* and *proxies*. An access structure consists of one manager node and any number of proxy nodes (which may increase or decrease during execution). An access structure is shaped like a star, with the manager at the center, and referenced by proxies. On each site, the proxy is the local reference to the access structure.

Legal transformations of the distribution graph must correspond to legal transformations of the language graph. Both must correspond to legal OPM reductions. These conditions on the node objects in each graph reflect the requirement of network transparency.

6.1.3 Mobility in the distribution graph

At this point, it is useful to clarify how cell mobility fits into the distribution graph model. First, the nodes of the distribution graph never change sites. A manager node has a global name that is unique across the network and never changes. This makes memory management very simple, as explained in Section 6.3.3. Second, access structures can move across the network (albeit slowly) by creating proxies on fresh sites and by losing local references to existing proxies. Third, the cell's content-edge (its state pointer) is known by exactly one proxy, i.e., it is localized to a single site. The content-edge can change sites (quickly) if requested to do so by a remote exchange. This is implemented by a change of state in the cell proxies that is coordinated by the mobility protocol.

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

It is sometimes necessary for a proxy to change managers. For example, object movement within Belgium is expensive if the manager is in Sweden. We assume that changing managers will be relatively infrequent. We can therefore use a more expensive protocol. For example, assume the old manager knows all its proxies. Then it sends messages to each proxy informing it of its new manager. This guarantees that eventually each proxy registers itself to the new manager. The protocol to change managers has been designed but not yet implemented.

6.1.4 The failure model

Failure detection must reliably inform the programmer if and when a failure occurs. A failure due to distribution appears in the language as an exception. This section summarizes the current design of how to extend the cell mobility protocol to provide precise failure detection. Since this issue is still under discussion, the final design may differ.

We distinguish between network failure and site failure. The current failure model handles site failure only. Either kind of failure becomes visible lazily in a proxy. When the proxy attempts to do a cell operation, then it finds out that there has been a failure. The proxy then becomes a failure node and any further messages to it are ignored. An exception is raised in the thread that initiated the cell operation.

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

- **Complete access structure failure**. This happens if either the manager node fails or a proxy that may contain the content-edge fails. The manager does not know at all times precisely where the content-edge is. The manager can bound the set of proxies that may contain the content-edge (see [28]). The manager knows that the content-edge is somewhere in this bounded set. If one proxy in the chain fails, then complete access structure failure is assumed to occur.
- **Proxy failure**. This happens if a proxy fails that is not in the chain. This does not affect the computation and may be safely ignored.

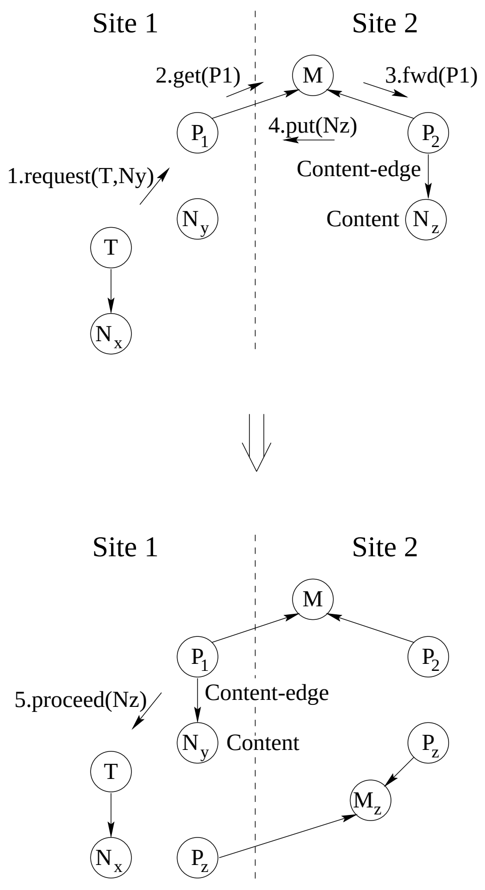

Figure 8: Exchange initiates migration of content-edge

6.2 The mobility protocol

This section gives an informal presentation of the mobility protocol for cells and ports. The protocol is very similar to a cache coherence protocol, as used in shared-memory multiprocessing. See [28] for a formal definition and a proof that the protocol is network transparent, i.e., that it is an implementation of the language semantics.

6.2.1 Cell mobility

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

The content-edge is stored at one of the cell proxies. Cell proxies exchange messages with threads in the engine. To ask for the cell content, a thread sends a message to a proxy. The thread then blocks waiting for a reply. After executing its protocol, the proxy sends a reply giving the content. This enables the thread to do the binding. Figure 8 shows how this works. We assume that the content-edge is not at the current proxy. A proxy requests the content-edge by sending a message to the manager. The manager serializes possible multiple requests and sends forwarding commands to the proxies. The current location of the content-edge may lag behind the manager's knowledge of who is the eventual owner. This is all right: the content-edge will eventually be forwarded to every requesting site.

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

Messages. P_i denotes the addresses of a proxy in the distribution graph corresponding to cell `C`. N_x, N_y, N_z denote the addresses of nodes corresponding to variables `X`, `Y`, and `Z`. A manager understands **get(P)**. A proxy understands **put(N)**, **fwd(P)** and **request(T,N)**, where T is the requesting thread. A thread understands **proceed(N)**.

Outline of protocol. (see Figure 8)

1. Proxy P_1 receives a **request(T,N** $_y$**)** from the engine. This message is sent by thread T as part of executing `{Exchange C X Y}`. Thread T blocks until the proxy replies. N_y is stored in P_1 (but does not yet become the content-edge). If the content-edge is at P_1, then P_1 immediately sends **proceed(P** $_z$**)** to T. Otherwise, **get(P** $_1$**)** is sent to the manager.

2. Manager M receives **get(P** $_1$**)**. Manager sends **fwd(P** $_1$**)** to the current owner P_2 of the content-edge, and updates the current owner to be P_1.

3. Proxy P_2 receives **fwd(P** $_1$**)**. If P_2 has the content-edge, which points to N_z, then it sends **put(N** $_z$**)** to P_1 and invalidates its content-edge. Otherwise, the content-edge will eventually arrive at P_2. The message **put(N** $_z$**)** causes the creation of a new access structure for N_z, for all entities N_z except records. For the access structure N_z is converted to P_z.

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

5. Thread T receives **proceed(N** $_z$**)**. The thread then invokes the binding of N_x and N_z.

6.2.2 Port mobility

The port protocol is an extension of the cell protocol defined in the previous section. As explained in Section 4.2.1, a port has two operations, send and localize, which are initiated by a thread referencing the port. The localize operation uses the same protocol as the cell exchange operation. For a correct implementation of the send operation, the port protocol must maintain the FIFO order of messages even during port movement. Furthermore, the protocol is defined so that there are no dependencies between proxies when moving a port. This means that a single very slow proxy cannot slow down a localize operation.

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

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

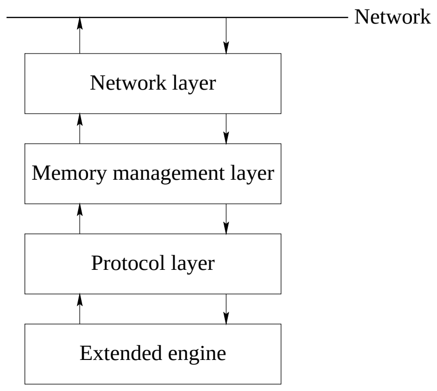

Figure 9: The implementation architecture

6.3 The implementation architecture

The implementation architecture is given in Figure 9. More information can be found in [28]. Here we give a brief summary of the functionality of each layer.

6.3.1 Extended engine

The engine extends the centralized Oz implementation with an interface to the protocol layer. The engine recognizes certain events and passes them to the protocol layer. A typical example is binding a variable through its proxy. Inversely, the protocol layer can initiate operations in the engine. Two typical examples are the arrival of the content of a cell from another site and passing a procedure from the network to the engine.

6.3.2 Protocol layer

The nodes of the distribution graph are modeled as concurrent objects. These objects exchange messages along the edges of the graph. The algorithms controlling these messages are known as the *distribution protocols* of the implementation. These protocols define the behavior of the system. Each access structure has its own distribution protocol. These protocols are summarized as follows:

- **Variable access structure**. This represents a globalized variable. The proxies are local representatives of the variable on each site. The proxies go away (i.e., become references, like local variables) when the variable access structure is merged into another access structure. The manager's sole purpose is to wait for the first merge request and then to tell all proxies to change managers. The set of proxies forms a multicast group. This protocol is the basic primitive used to add information to the constraint store.

- **Procedure access structure**. This represents a globalized procedure, which is stateless and therefore can be copied to other sites (*replicated*) when needed there. The protocol ensures that each closure or code block of a procedure is unique on a site. The manager handles requests for copies. The manager is associated with a local node representing the structure or procedure.

- **Mobility access structure**. This represents a cell or port. The state is localized, that is, the content-edge is always on exactly one site. The content-edge can move from one proxy to another. A proxy that needs the content-edge will ask the manager. The manager decides who gets the content-edge next, and sends a forwarding request to the proxy that will eventually get the content-edge. Section 6.2 gives an informal presentation of this protocol.

6.3.3 Memory management layer

The memory management layer translates the distribution graph structure into byte sequences to be transferred across the network. This translation is intimately tied with the building of access structures and the reclamation of their nodes that are no longer accessible.

- The messages exchanged between nodes of the distribution graph may reference subgraphs (e.g., large data structures). When subgraphs are transferred from one site to another, their nodes must become members of access structures that have nodes both on the sender and receiver site.

- Access structures are identified by network-wide unique addresses, called *network addresses*. A network address is reclaimed when the access structure that it identifies is no longer locally accessible on any of its sites. This is detected by using the local garbage collectors together with a credit mechanism. Each network address is created with a fixed large number of credits. The manager initially owns the credits and gives credits to any site (including messages in transit) that has the network address. When a site no longer locally references the access structure then the credits are sent back to the manager. When the manager recovers all its credit and is no longer locally referenced, then it is reclaimed.

6.3.4 Network layer

The network layer implements a cache of TCP connections to provide reliable transfer between arbitrary sites on a wide-area network [4]. Recent implementations of TCP can outperform UDP [2]. This layer implements a cache of TCP connections. We assume a reliable network with no bounds on transfer time and no assumptions on relative ordering of messages (no FIFO) for all distribution protocols except the stationarity access structure. To send arbitrary-length messages from fair concurrent threads, the implementation manages its own buffers and uses non-blocking send and receive system calls.

7 Status and future work

Distributed Oz is a conservative extension to Oz for distributed programming. The prototype implementation incorporates all of the ideas presented in this article, including mobile objects with predictable network behavior and support for open computing. The prototype is an extension of the centralized Oz 2.0 system and has been developed jointly by the Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI) in the Perdio project [25] and by the Swedish Institute of Computer Science. Oz 2.0 is a publicly-available system with a full-featured development environment [19]. The original DFKI Oz system [19], a robust and efficient implementation of Oz, has been available since Jan. 1995. Oz provides a sophisticated concurrent object system as well as constraint programming facilities.

Current work includes improving the efficiency and robustness of the prototype, using it in actual applications, and building the standard services needed for distributed application development. Future work includes adding fault tolerance based on precise failure detection, distributed exception handling, and persistence, adding support for resource management and multiprocessor execution (through "virtual sites") and adding support for security.

Acknowledgements

We thank our colleagues at SICS and DFKI for their contributions, and especially Per Brand. This research is supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FTZ-ITW-9105) and the Esprit Working Group CCL (EP 6028). This research is funded in Sweden by the Swedish national board for industrial and technical development (NUTEK) and SICS.

References

[1] Tomas Axling, Seif Haridi, and Lennart Fahlen. Virtual Reality Programming in Oz. Feb. 1996.

[2] Brent Callaghan. *WebNFS–The file system for the World-Wide Web*. SunSoft, Inc., May 1996. Available at `http://www.sun.com/solaris/networking/webnfs/`.

[3] Luca Cardelli. *Obliq: A Language with Distributed Scope*, DEC Systems Research Center, Research Report, Nov. 1994.

[4] Douglas E. Comer. *Internetworking with TCP/IP Volume I: Principles, Protocols, and Architecture*, Prentice Hall, 1995.

[5] K. Fischer and N. Kuhn and J. P. Müller. Distributed, Knowledge-Based, Reactive Scheduling in the Transportation Domain. Proc. 10th IEEE Conf. on Artificial Intelligence and Applications, San Antonio, Texas, March 1994.

[6] K. Fischer, J. P. Muller, and M. Pischel. A Model for Cooperative Transportation Scheduling. In Proc. 1st Int. Conf. on Multiagent Systems (ICMAS '95). San Francisco, June 1995, pages 109–116.

[7] Olof Hagsand. DIVE–A Platform for Multi-User Virtual Environments. In *IEEE Multimedia*, Spring 1996.

[8] Seif Haridi. *An Oz 2.0 Tutorial*. Available at `http://sics.se/~seif/oz.html`.

[9] Martin Henz, Martin Müller, and Markus Wolf. *Munchkins: A shell for distributed multi-user games.* In *WOz'95, International Workshop on Oz Programming*, Institut Dalle Molle d'Intelligence Artificielle Perceptive, Martigny, Switzerland, Nov. 1995. Available at `http://ps-www.dfki.uni-sb.de/~henz/oz/munchkins/`.

[10] Martin Henz, Gert Smolka, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In V. Saraswat and P. Van Hentenryck, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 29–48. The MIT Press, Cambridge, MA, 1995.

[11] Martin Henz and Jörg Würtz. Using Oz for College Timetabling. In Int. Conf. on the Practice and Theory of Automated Timetabling, Edinburgh, Scotland, Aug. 1995.

[12] Martin Henz, Stefan Lauer, and Detlev Zimmermann. COMPOzE–Intention-based Music Composition through Constraint Programming. In Proc. IEEE Int. Conf. on Tools with Artificial Intelligence, Nov. 1996, Toulouse, France.

[13] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. In *TOCS*, Vol. 6, No. 1, Feb. 1988, pages 109–133.

[14] Lawrence Berkeley Laboratory. Whiteboard tool "WB". Available at `http://mice.ed.ac.uk/mice/archive`.

[15] Lone Leth and Bent Thomsen. *Some Facile Chemistry*, European Computer-Industry Research Centre, Report ECRC–92–14, May 1992.

[16] General Magic, Inc. *Telescript Developer Resources*, Available at `http://www.genmagic.com/Develop/Telescript/tsdocs.html`.

[17] Peter Magnusson. *Internet Perspectives*. Presentation at Memtek seminar, March 1996. Available at `http://www.sics.se/~psm/internet_perspectives/`.

[18] Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA: The Common Object Request Broker Architecture*. Prentice Hall PTR, 1996.

[19] Programming Systems Lab, DFKI. *The DFKI Oz system version 1.1*, German Research Centre for Artificial Intelligence (DFKI), 1995. Available at `http://ps-www.dfki.uni-sb.de/`.

[20] Vijay A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, Cambridge, MA, 1993.

[21] S. Schmeier and Schupeta Achim. PASHA II — Personal Assistant for Scheduling Appointments. Proc. 1st Int. Conf. on the Practical Application of Intelligent Agents and Multi-Agent Technology, 1996.

[22] Gert Smolka. *The Oz Programming Model*, In *Computer Science Today*, Springer LNCS 1000, 1995, pages 324–343.

[23] Gert Smolka. *An Oz Primer*. In *DFKI Oz Documentation*, 1995. Available at `http://ps-www.dfki.uni-sb.de/oz/documentation/#crash`.

[24] Gert Smolka. *The Definition of Kernel Oz*, In *Constraints: Basics and Trends*, Springer LNCS 910, 1995, pages 251–292.

[25] Gert Smolka, Christian Schulte, and Peter Van Roy. *PERDIO–Persistent and Distributed Programming in Oz*, BMBF project proposal, Feb. 1995. Available at `http://ps-www.dfki.uni-sb.de/fbps/perdio.html`.

[26] W. Richard Stevens. *Remote Procedure Calls*, Chapter 18 of *Unix Network Programming*, Prentice Hall Software Series, 1990.

[27] Sun Microsystems, Inc. *The Java Series*. Addison-Wesley, 1996. Available at `http://www.aw.com/cp/javaseries.html`.

[28] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. *Mobile Objects in Distributed Oz*. Submitted to ACM TOPLAS, Jan. 1997.

[29] Joachim P. Walser. Feasible Cellular Frequency Assignment Using Constraint Programming Abstractions. In Proc. 1st Workshop on Constraint Programming Applications, CP-96, Cambridge, MA, 1996.

[30] Claes Wikström. *Distributed Programming in Erlang*, First International Symposium on Parallel Symbolic Computation (PASCO '94), World Scientific, Sep. 1994, pages 412–421.