Mobile Objects in Distributed Oz

PETER VAN ROY
Université Catholique de Louvain
SEIF HARIDI and PER BRAND
Swedish Institute of Computer Science
and
GERT SMOLKA, MICHAEL MEHL, and RALF SCHEIDHAUER
German Research Center For Artificial Intelligence (DFKI)

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

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*; *data-flow languages*; *object-oriented languages*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*operational semantics*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Mobile objects, network transparency, latency tolerance

The development of Distributed Oz at DFKI is supported by the BMBF through Project PERDIO (FKZ ITW 9601). This research is funded in Sweden by the Swedish national board for industrial and technical development (NUTEK) and SICS.

Author's addresses: P. Van Roy, Department of Computing Science and Engineering, Université Catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium; email: pvr@info.ucl.ac.be; S. Haridi and P. Brand, Swedish Institute of Computer Science, S-164 28 Kista, Sweden; email: {seif; perbrand}@sics.se; G. Smolka, M. Mehl, and R. Scheidhauer, German Research Center for Artificial Intelligence, D-66123 Saarbrücken, Germany; email: {smolka; mehl; scheidhr}@dfki.de.

1. INTRODUCTION

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

1.1 Object Mobility

Making mobility a primitive concept makes it possible to define efficient networked objects whose mobility can be precisely controlled (see Section 5.4.2). The object can change sites on its own or on request. The object does not leave a trail, i.e., it does not leave behind aliases or surrogate objects to forward messages when it changes sites. There is no "proxy explosion" problem when an object is passed repeatedly between two sites [Foody 1997]. Many sites may send messages to the object. It is eventually true that messages sent will go to the object in a single network hop, no matter how many times the object moves. There is no difference in syntax or computational behavior between these objects and stationary objects. No published system has objects with these abilities. In particular, Emerald [Jul et al. 1988], Obliq [Cardelli 1995], and Java with Remote Method Invocation [Sun Microsystems 1996] all suffer from the aliasing problem to some extent. One of the contributions of this article is to show how to provide mobile objects with predictable network behavior, i.e., without aliasing, in a simple and straightforward way.

1.2 Two Semantics

The basic design principle of Distributed Oz is to distinguish clearly between the *language* semantics and the *distributed* semantics. Distributed Oz has the same language semantics as Oz 2, a concurrent object-oriented language that is state aware and has dataflow synchronization. The object system has a simple formal foundation and yet contains all the features required in a modern concurrent language. Detailed information about the language and its object system can be found in Haridi [1996] and Henz [1997].[1] Implementations of Oz and its successor Oz 2 have been used in many research projects [Axling et al. 1995; Carlsson and Hagsand 1996; Fischer et al. 1994, 1995; Henz and Würtz 1996; Henz et al. 1996; Schmeier and Achim 1996; Walser 1996]. To be self-contained, this article uses a subset of Oz 2 syntax that directly corresponds to its semantics.

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

The distributed semantics extends the language semantics to take into account the notion of site. It defines the network operations invoked when a computation is partitioned on multiple sites. There is no distribution layer added on top of an existing centralized system. Rather, all language entities are given a network behavior that respects the same language semantics they have when executing locally. By a *language entity* we mean a basic data item of the language, such as an object, a procedure, a thread, or a record. Figure 3 classifies the entities and summarizes their distributed semantics. Network operations[2] are predictable, which gives the programmer the ability to manage network communications.

1.3 Developing an Application

Developing an application is separated into two independent parts. First, the application is written without explicitly partitioning the computation among sites. One can in fact check the *safety* and *liveness* properties[3] of the application by running it on one site. Second, the application is made *efficient* by controlling the mobility of its entities. For example, some objects may be placed on certain sites, and other objects may be given a particular mobile behavior. The shared graphic editor of Section 2 is designed according to this approach.

1.4 Mobility Control and State

The distributed semantics extends the language semantics with *mobility control.* In general terms, mobility control is the ability for stateful entities to migrate between sites or to remain stationary at one site, according to the programmer's intention [Haridi et al. 1997]. The programmer can use mobility control to program the desired network communication patterns in a straightforward way. For example, to reduce network latency a mobile object can behave as a state cache. This is illustrated by the shared graphic editor of Section 2.

By *stateful entities* we mean entities that change over time, i.e., they are defined by a sequence of states, where a *state* can be any entity. At any given moment, a stateful entity is localized to a particular site, called its *home* site. Stateful entities are of two kinds, called *cells* and *ports*, that are respectively mobile and stationary. Objects are defined in terms of cells. The mobility behavior of an object is defined in terms of cells and ports. The language semantics of these entities is given in Section 4, and their distributed semantics is given in Section 5. The implementation contains a mobile state protocol that implements the language semantics of cells while allowing the cell's state[4] to efficiently migrate between sites.

It is important that *all* language entities have a well-defined network behavior. For example, the language provides network references to procedures. A procedure is stateless, i.e., its definition does not change over time. Calling the procedure locally or remotely gives the same results. Disregarding sites, it behaves identically to a centralized procedure application. Passing a procedure to a remote site causes a network reference to be created to the procedure. Calling the procedure causes it to be replicated to the calling site.[5] The procedure's external references will be

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

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

[4] More precisely, its *content-edge*, which is defined in Section 4.1.

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

either replicated or remotely referenced, depending on what kind of entity they are. Only the first call will have a network overhead if the procedure does not yet exist on the site.

1.5 Overview of the Article

This article consists of eight parts and an appendix. Section 2 presents an example application, a shared graphic editor, that illustrates one way to use mobile objects to reduce network latency. Section 3 lays the foundation for our design by reasoning from four general requirements for distributed programming. Section 4 summarizes the language semantics of Distributed Oz in terms of these requirements. Section 5 defines and justifies the distributed semantics of Distributed Oz. We show by example how easy it is to code various kinds of migratory behavior using cells and ports. Section 6 outlines how mobility is introduced into the language semantics and specifies a mobile state protocol for cells. Section 7 summarizes the system architecture and situates the protocol in it. Section 8 compares the present design with distributed shared memory, Emerald, and Obliq. Section 9 summarizes the main contributions and the status of the project. Finally, Appendix A gives a formal proof that the mobile state protocol implements the language semantics for cells.

2. A SHARED GRAPHIC EDITOR

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

Figure 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 using the open

must be introduced.

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

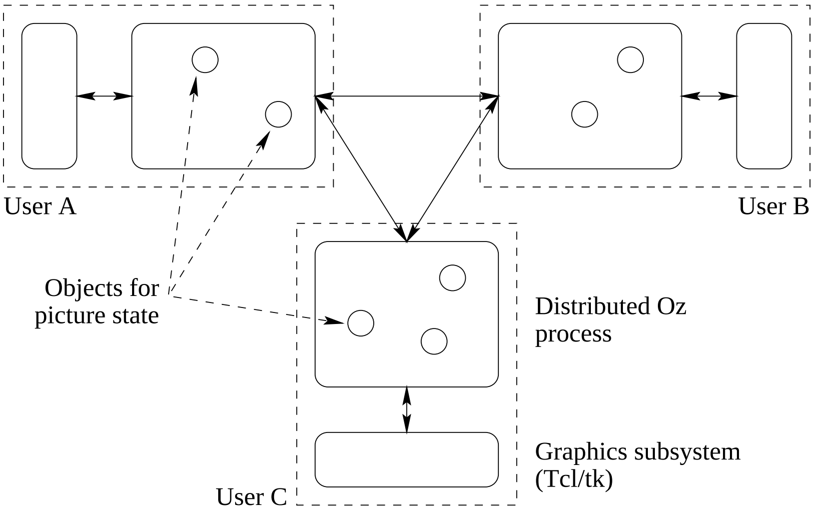

Fig. 1. A shared graphic editor.

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

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

To solve the latency problem, we refine the design to represent the picture state and the distributed agenda as freely mobile objects rather than stationary objects. The effect of this refinement is that parts of the picture state are cached at sites that modify them. Implementing the refinement requires changing some of the calls

Table I. System Requirements and Some of their Mechanisms

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

that create new objects. In all, less than 10 lines of code out of 500 have to be changed. With these changes, freehand sketches do not need any network operations to update the local display, so performance is satisfactory. Remote users see the sketch being made in real time, with a delay equal to the network latency.

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

In general, mobile objects are useful both for fine-grain mobility (caching of object state) as well as coarse-grain mobility (explicit transfer of groups of objects). The key ability that the system must provide is transparent control of mobility, i.e., control that is independent of the object's functionality. Section 5.4 shows how this is done in Distributed Oz.

3. LANGUAGE PROPERTIES

In order to provide a firm base for the language design, we start from four requirements that are generally agreed to be important in a distributed setting. We then propose a set of mechanisms that are sufficient to satisfy these requirements. The four requirements are network transparency, flexible network awareness, latency tolerance, and language security. Table I summarizes the requirements and their enabling mechanisms. Section 4 presents a design that contains all the mechanisms. For brevity, we give only a summary of the fault model. Other important requirements such as resource management and network security will be presented elsewhere.

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

3.1 Network Transparency

Network transparency means that computations behave in the same way independent of the distribution structure.[7] That is, the language semantics is obeyed independent of how the computation is partitioned onto multiple sites. This requires a *distributed shared computation space*, which provides the illusion of a single networkwide address space for all entities (including threads, objects, and procedures). The distinction between local references (on the same site) and remote references (to another site) is invisible to the programmer. Consistency of remote access is explained in Section 5. For reasons of security, the computation space is not just a shared memory. This is explained below.

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

3.2 Flexible Network Awareness

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

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

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

3.3 Latency Tolerance

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

[7] The terms "network transparency" and "network awareness" were first introduced by Cardelli [1995].

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

one thread waits for the network, other threads can continue.

Caching improves latency tolerance by increasing the locality of computations. Caching stateless entities amounts to replicating them. Caching stateful entities requires a coherence protocol. The mobile state protocol of Distributed Oz guarantees coherence and is optimized for frequent state updates and moves. The current design does not yet have a replacement policy, i.e., there are no special provisions for resource management.

Dataflow synchronization allows computations to stall only on data dependency (not on send or receive). Well-known techniques to achieve this are *futures* [Halstead 1985], *I-structures* [Iannucci 1990], and *logic variables* [Bal et al. 1989; Shapiro 1989]. We use logic variables because they have great expressive power, are easily implemented efficiently, and are consistent with the semantics of a state-aware language (see Section 4.1.2). *Asynchronous ordered communication* generalizes logic variables by adding a form of buffering. This is provided by *ports* (see Section 4.2.2).

3.4 Language Security

Language security means that the language guarantees integrity of computations and data. It is important to distinguish between *language* security and *implementation* security. Implementation security means that integrity of computations is protected against adversaries that have access to the system's implementation. This is beyond the scope of the article, though. We provide language security by giving the programmer the means to restrict access to data. Data are represented as references to entities in an abstract shared computation space. The space is *abstract* because it provides a well-defined set of basic operations. In particular, unrestricted access to memory is forbidden.[9] One can only access data to which one has been given an explicit reference. This is controlled through lexical scoping and first-class procedures [Abelson et al. 1985]. *Lexical scoping* means that a program's initial references are determined by its static structure. Other references are passed around explicitly during execution. *First-class procedures* means that procedures can be created and applied dynamically and that references to them can be passed around:

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

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

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

Table II. Summary of OPM Syntax

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

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

4. LANGUAGE SEMANTICS

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

Distributed Oz is defined by transforming all its statements into statements of a small kernel language, called OPM (Oz Programming Model) [Smolka 1995a; Smolka 1995b]. OPM is a concurrent programming model with an interleaving semantics. It has three innovative features. First, it uses dataflow synchronization through logic variables as a basic mechanism of control. Second, it makes an explicit distinction between stateless references (logic variables) and stateful references (cells). Finally, it is a unified model that subsumes higher-order functional and object-oriented programming.

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

4.1 Oz Programming Model

This section summarizes OPM, the formal model underlying Oz 2. Readers interested mainly in the object system may skim directly to Section 4.2 on first reading. A program written in OPM consists of a (compound) statement containing value descriptions, variable declarations, procedure definitions and calls, state declarations and updates, conditionals, thread declarations, and exception handling (see Table II).

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

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

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

A *name* has no external representation and hence cannot be forged within the language. The call `{NewName Z}` creates a new name that is unique systemwide. Names are used to identify cells, procedures, and threads. Names can be used to add hidden functionality to entities. For example, the server loop of Section 5.4.2 is stopped with a secret name. Names are to language security what capabilities are to implementation security.

4.1.2 *Variable Declaration.* All variables are logic variables. They must be declared in an explicit scope bracketed by **local** and **end**. The system enforces that a variable always refers to the same value. A variable starts out with its value unknown. The value becomes known by *binding* the variable, i.e., after executing the binding `X=V`, variable `X` has value `V`.[10] The binding operation is called *incremental tell* [Smolka 1995b]. In essence, incremental tell only adds variable bindings that are consistent with existing bindings in the store.

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

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

```
local X in                 % Declare X, value is unknown
```

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

```
   thread {Consumer X} end % Create thread, use X
   {Producer X}            % Calculate value of X
end
```

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

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

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

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

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

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

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

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

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

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

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

```
local Loop in
   proc {Loop N}   % Define a procedure
      {Loop N+1}
   end
   thread          % Run an infinite loop in a new thread
      {Loop 0}
   end
end
```

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

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

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

The **try** S_1 **catch** E **then** S_2 **end** defines a context for exception handling in the current thread. If an exception `T` is raised during the execution of S_1 in the same thread then control will transfer to the **catch** clause of the innermost **try**. If

T matches E then S_2 is executed, and the **try** is exited; otherwise the exception is reraised at an outer level. User-defined exceptions can be raised by the statement **raise** T **end**.

4.2 Compound Entities

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

- —Concurrent objects with explicit reentrant locking. There is syntactic support for classes with multiple inheritance and late binding.
- —Ports, which are asynchronous channels related to *M-structures* [Barth et al. 1991].

These entities are entirely defined in terms of OPM. They are provided because they are useful abstractions. In Section 5 they are given a specific distributed semantics.

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

Class definitions and object declarations are both executed at run-time. A class definition builds the method table and resolves conflicts of multiple inheritance. Like OPM, both class definitions and object declarations are fully compositional with respect to all language features. For example, arbitrary nesting is allowed between class, object, procedure, and thread declarations. The following presents a definition of objects consistent with Oz 2. For more information see Smolka [1995a], Haridi [1996], Henz [1997], and Smolka et al. [1995].

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

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

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

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

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

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

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

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

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

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

$} is equivalent to {NewLock Lock}. The dollar symbol is used as a placeholder. We use the notation **proc** {$ ...} ... **end** for anonymous procedures.

The procedure {NewLock Lock} returns a reentrant lock named Lock. In OPM a reentrant lock is a procedure that takes another procedure P as argument and executes P in a critical section. The lock is thread-reentrant in the sense that it allows the same thread to enter other critical sections protected by the same lock. Other threads trying to acquire the lock will wait until P is completed. The definition of NewLock in OPM will be given shortly. As we will see, thread-reentrant locking is modeled in OPM using cells and logic variables.

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

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

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

This assumes that each thread has a unique identifier T that is different from the literal **unit** and that is obtained by calling the procedure {GetThreadID T}. The

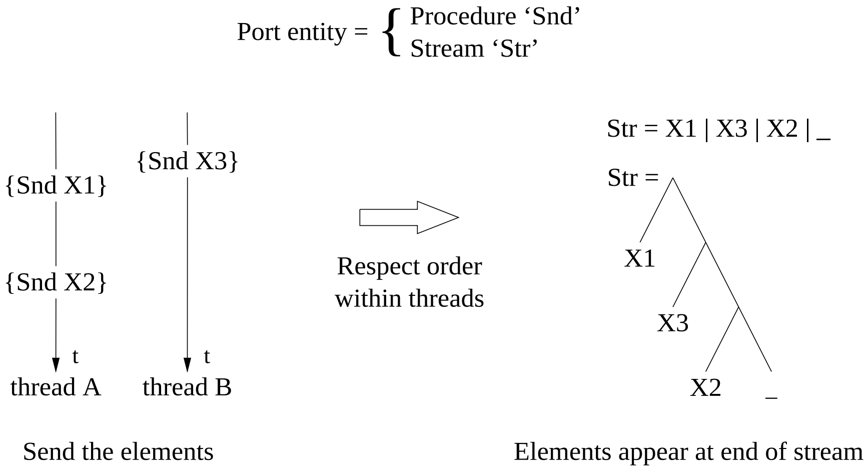
Port entity = { Procedure 'Snd' Stream 'Str'
{Snd X1}
{Snd X2}
{Snd X3}
t
t
thread A
thread B
Send the elements
Respect order within threads
Str = X1 | X3 | X2 | _
Str =
X1
X3
X2
_
Elements appear at end of stream

Fig. 2. Ports: asynchronous channels with multicast ability.

`{Wait Old}` call blocks the thread until `Old`'s value is known. The **`try ... finally`** S **`end`** is syntactic sugar that ensures S is executed in both the normal and exceptional cases, i.e., an exception will not prevent the lock from being released.

4.2.2 *Ports.* A *port* is an asynchronous channel that supports ordered many-to-one and many-to-many communication. A port consists of a send procedure and a stream (see Figure 2). A *stream* is a list whose tail is a logic variable. Sends are asynchronous and may be invoked concurrently. The entries sent appear at the end of the stream. The send order is maintained between entries that are sent from within the same thread. No guarantees of order are given between threads.

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

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

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

Calling `{NewPort Str Snd}` creates the procedure `Snd` and ties it to the stream `Str`. Calling `{Snd X}` appends `X` to the stream and creates a new unbound end of the stream. That is, the tail `S` of the stream is bound (`S=X|S2`), and `S2` becomes

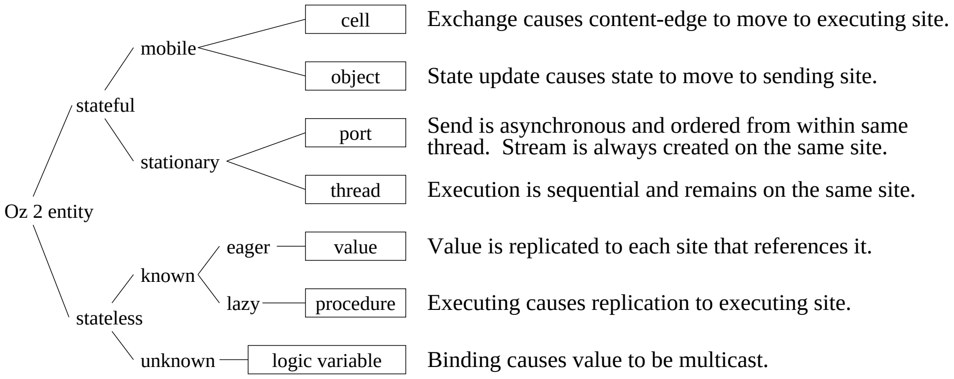
Oz 2 entity
stateful
stateless
mobile
stationary
known
unknown
eager
lazy
cell
object
port
thread
value
procedure
logic variable
Exchange causes content-edge to move to executing site.
State update causes state to move to sending site.
Send is asynchronous and ordered from within same thread. Stream is always created on the same site.
Execution is sequential and remains on the same site.
Value is replicated to each site that references it.
Executing causes replication to executing site.
Binding causes value to be multicast.

Fig. 3. Oz 2 entities and their distributed semantics.

the new tail. One way to build a *server* is to create a thread that waits until new information appears on the stream and then takes action, depending on this information. Section 5.4.1 shows how to use a port to code a server.

5. DISTRIBUTION MODEL

Distributed Oz defines a distributed semantics for all basic entities and compound entities. By this we mean that each operation of each entity is given a well-defined network behavior. This section defines and motivates this behavior. The distributed semantics for all seven entities is classified and summarized in Figure 3. We discuss separately replicable entities (values and procedures), logic variables, and stateful entities (cells, objects, ports, and threads). The semantics of the basic stateful entity, the cell, are defined in Section 6. Appendix A proves correctness of the cell's mobile state protocol. Definitions and correctness proofs for objects and ports should be easy to devise after understanding the cell.

5.1 Replication

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

We summarize the *lazy replication* protocol, i.e., the distributed algorithm that manages replication. Records, numbers, and literals are replicated eagerly, i.e., there is no such thing as a remote reference to a record.[11] Procedures are replicated lazily, i.e., one may have remote references to a procedure. The procedure is replicated when it is applied remotely. Both the compiled code and the closure are given global addresses. Therefore a site has at most one copy of each code block and

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

closure. All network messages, except for responses to explicit replication requests, do not contain any procedure code nor closure. A replication request is sent only if the code or closure is not available locally.

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

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

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

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

5.2 Logic Variables

A logic variable is a reference to a value that is not yet known. Since a logic variable does not correspond to a sequence of values, but to a single value, we consider it to be stateless. There are two basic operations on a logic variable: binding it and waiting for it to have a value. Binding a logic variable eagerly replaces the logic variable by its value on all sites that reference it. Since a value is stateless, any number of readers can wait concurrently for the value to become known. This binding protocol, also called *variable elimination* protocol, is the only distributed algorithm needed to implement distributed unification.[12]

In addition to their role in improving latency tolerance, logic variables provide the programmer with an efficient and expressive way to dynamically manage multicast groups. A *multicast* sends data to a predefined subset of all network addresses, called the multicast group. Recent protocol designs support multicast as a way to increase network efficiency [Deering 1989]. Binding a logic variable can be implemented using a multicast group. Binding to a record multicasts the record. Binding to a procedure multicasts only the name of the procedure (not the closure). Binding to another logic variable merges the two multicast groups.

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

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

5.3 Mobility Control

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

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

— **Cells.** A cell is mobile. A cell may be accessible from many sites, each of which knows the cell name. Only the home site contains the cell's content-edge, which pairs the name and the content. Invoking an exchange from any site causes a synchronous move of the content-edge to that site. This is done using a mobile state protocol to implement the interleaving semantics of the exchanges (see Section 6). Invoking a cell-access operation does not move the content-edge. Only the cell's content is transferred to the invoking site.

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

— **Ports.** A port is stationary. Invoking a send from any site causes a new entry to appear eventually in the port's stream at its home site. The send operation is defined to be *asynchronous* (nonblocking) and *ordered* (FIFO). This behavior cannot be defined in terms of OPM. It is proper to the distributed semantics. Send operations complete immediately (independently of any network operations), and messages sent from a given thread appear on the stream in the same order that they were sent.

We provide ports as an entity for two reasons. First, a stationary port is a natural way to build a server. Second, the asynchronous ordering supports common programming techniques and can exploit popular network protocols. Ports have a second operation, localize, which causes a synchronous move of the home site to the invoking site. Without the ability to localize ports, mostly stationary objects cannot be implemented transparently (see Section 5.4.2).

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

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

Fig. 4. Making an object stationary (first attempt).

thread access were packaged in a port, and the calling thread suspends until the operation is performed.

The protocols used to implement mobility control for objects and ports are both based on the mobile state protocol given in this article. They are extended to provide for the operations of the particular entity. For example, the port protocol manages the mobility of the home site as well as the FIFO connections from other sites to the home site. The port protocol is defined in Section 6.4.4.

5.4 Programming with Mobility Control

We show how to concisely program arbitrary migratory behavior using the entities of Distributed Oz. Expressing other distributed-programming idioms (e.g., RPC and client-server architectures) is left as an exercise for the reader. We assume the existence of primitives to initiate a computation on a new site.

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

—**Freely mobile objects.** This is the default behavior for objects. Any object defined as in Section 4.2.1 will move to each site that sends a message that updates the object's state. The object is guaranteed to stay at the site until the lock is released within the invoked method. While the lock is held, the object will not move from the site.

—**Stationary objects.** A stationary object executes all its methods on the same site. Any object can be made stationary using the technique given in Section 5.4.1.

—**Mostly stationary objects.** A mostly stationary object remains stationary unless explicitly moved. Any freely mobile object can be made mostly stationary using the technique given in Section 5.4.2.

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

Fig. 5. Making an object stationary (correct solution).

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

5.4.1 *Stationary Objects.* An object can be made stationary be wrapping it in a port. Figure 4 shows a simple way to do this by defining the procedure `MakeStat` that takes any object `Obj` and returns the procedure `StatObj`. The result of calling `StatObj` is to send messages to a port. The thread in the construction **`thread`** `{Serve Str}` **`end`** is responsible for the actual object invocation. The thread takes messages from the port's stream and sends them to the object. The thread does not move. Therefore, `StatObj` behaves like `Obj` except that `Obj` does not move. After the first object invocation, the object is at the site of the server loop and will not move (unless, of course, some threads are given direct references to `Obj`). For example, if upon its creation `Obj` is passed directly to `MakeStat`, and only references to `StatObj` are passed to others, then `Obj` will forever remain on its creation site.

However, the solution in Figure 4 is too simple. `StatObj` deviates from providing *exactly* the same behavior as `Obj` in three ways. First, sending messages to `StatObj` is asynchronous, whereas sending messages to `Obj` is synchronous. Second, only one method of `Obj` can be executing at a time, since `Obj` is inside its own thread. Third, exceptions in `Obj` are not seen by `StatObj`.

Figure 5 gives a correct definition of the procedure `MakeStat`. The logic variable `Sync` is used to synchronize `{StatObj M}` with `{Obj M}`. The notation `msg(M Sync)` pairs `M` and `Sync` in a record. Waiting until `Sync` is bound to a value guarantees that the thread executing `{StatObj M}` continues execution only after `{Obj M}` is finished. This means that messages sent from within one thread are received in the order sent. The example also models exceptions correctly by transferring the exceptions back to the caller.[13]

There are many useful variations of this solution:

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

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

This solution makes it clear that a stationary object is not a simple concept. It requires synchronization between sites, passing of exceptions between sites, and thread creation. Freely mobile objects are simpler, since their execution is always local. The mechanics of making an object stationary can be encapsulated, as is done here, to hide its complexity from the user. In general, most of the complexity of concurrent programming can be encapsulated. The practicality of this approach is demonstrated on an industrial scale by the Ericsson Open Telecom Platform [Armstrong et al. 1996; Ericsson 1996].

5.4.2 *Mostly Stationary Objects.* It may be desirable in special cases to move an object that has been made stationary, e.g., for a server to move closer to its clients or to leave a machine that will be shut down. We require a solution in which it is eventually true that sending a message will need only a single network hop, no matter how many times the object moves. Figure 6 gives a solution that uses the localize operation for ports. Port mobility is derived from cell mobility; see Sections 5.3 and 6.4.4. The figure defines the procedure `MakeFirm` that takes any freely mobile object `Obj` and returns two procedures, `FirmObj` and `Move`. The procedure `FirmObj` has identical language semantics to `Obj` but is stationary.[14] `Move` is a zero-argument procedure that when applied, atomically moves the object to the invoking site. That is, in each thread's stream of messages to `FirmObj`, there is a point such that all earlier messages are received on the original site, and all later messages are received on the new site.

Like a stationary object, a mostly stationary object consists of an object wrapped in a port. To handle incoming messages, a server loop is installed on the port's home site. To move the object, this loop is stopped; the port's home site is moved using the localize operation; and a new server loop is installed on the new site. The server loop in Figure 6 is stopped by sending the message `Key(Stopped Rest)`,

[13] One subtle point remains that requires attention. The encapsulated object can escape by a method returning **`self`**. The problem can be solved by using inheritance to make the object stationary or by using the Oz 2 metaobject protocol [Henz 1997]. We do not show the details, since this would take us too deep into the object system.

[14] Adding the corrections of Figure 5 is left as an exercise for the reader.

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

Fig. 6. Making an object mostly stationary.

where `Key` is an Oz 2 name used to identify the message and where `Stopped` and `Rest` are outputs. Since `Key` is unforgeable and known only inside `MakeFirm`, the server loop can be stopped only by `Move`. The port `Prt` must be hidden inside a procedure; otherwise it can be localized by any client. When the loop is stopped, `Rest` is bound to the unprocessed remainder of its message stream. The new server loop uses `Rest` as its input.

From an algorithmic viewpoint, Figure 6 defines the distributed algorithm for mostly stationary objects. This algorithm is a composition of three simpler algorithms: (1) a mobile state protocol (for cells and extended for ports), which is the scope of this article, (2) a variable elimination protocol (see Section 5.2), and (3) a lazy replication protocol (see Section 5.1). The three algorithms are composed by means of a notation which is exactly the OPM language. The technique of factoring complex algorithms into simpler components has many advantages. For example, Figure 6 can be extended in a straightforward way with a failure model, using the exceptions of OPM.

6. CELLS: SEMANTICS AND MOBILE STATE PROTOCOL

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

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

6.1 Cell Semantics

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

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

$$\begin{array}{c||c} (\textit{statement}) & (\textit{new statement}) \\ \hline (\textit{store}) & (\textit{new store}) \end{array}$$

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

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

$$\textbf{Cell creation} \quad \begin{array}{c||c} \texttt{\{NewCell X C\}} & \texttt{C=n} \\ \hline \sigma & \sigma \wedge \texttt{n:X} \end{array} \quad \text{newName}(\texttt{n})$$

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

$$\textbf{Cell access} \quad \begin{array}{c||c} \texttt{\{Access C X\}} & \texttt{X=Z} \\ \hline \sigma \wedge \texttt{C=n} \wedge \texttt{n:Z} & \sigma \wedge \texttt{C=n} \wedge \texttt{n:Z} \end{array}$$

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

$$\textbf{Cell exchange}\quad \begin{array}{c||c} \texttt{\{Exchange C X Y\}} & \texttt{X=Z} \\ \hline \sigma \wedge \texttt{C=n} \wedge \texttt{n:Z} & \sigma \wedge \texttt{C=n} \wedge \texttt{n:Y} \end{array}$$

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

6.1.3 *Distributed Semantics of Exchange.* The distributed semantics is an extension of the language semantics that specifies how the reduction is partitioned among multiple sites. We introduce the notion of a *representative* on a site. This notion is used to place statements and the store contents on one or more sites. By store contents we mean content-edges and references to values or logic variables. The representative of X on site i is denoted by X_i. The subscript denotes the site. The exact notion of representative depends on what Oz 2 entity is considered (see Figure 3). In the case of cells it is defined here. For other entities it is straightforward to devise after understanding the language graph and distribution graph defined in Section 6.2.1. We define the distributed semantics in three steps:

—Define the representation of an entity as a formula written in terms of representatives.

—Define a mapping M from the representation of an entity to the entity it represents.

—Formulate the reduction rules in terms of representations.

To show that the distributed semantics implements the language semantics, we use a technique variously known as *abstraction* [Lampson 1993] or *simulation* [Lynch 1996]. Consider the configuration (S, σ). Its reduction gives a set of new configurations (S$'$, σ'). Consider the corresponding configuration (S_r, σ_r), written in terms of representatives. Its (distributed) reduction gives a set of (${\mathrm{S}'}_r$, ${\sigma'}_r$). This is summarized in the following diagram:

$$\begin{tikzcd}
(\mathrm{S},\sigma) \arrow[r, "ls"] & (\mathrm{S}',\sigma') \\
(\mathrm{S}_r,\sigma_r) \arrow[u, "\mathrm{M}"] \arrow[r, "ds"] & ({\mathrm{S}'}_r,{\sigma'}_r) \arrow[u, "\mathrm{M}"']
\end{tikzcd} \tag{1}$$

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

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

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

The cell is accessible from sites 1, ..., k, so that $\texttt{C}_1\texttt{=n}$, $\texttt{C}_2\texttt{=n}$, ... $\texttt{C}_k\texttt{=n}$ together imply `C=n`. The content-edge on site p is denoted by $(\texttt{n:Z})_p$. The other sites know the cell but do not have the content-edge. This is denoted by $(\texttt{n:}\bot)_i$ for $i \neq p$. If we

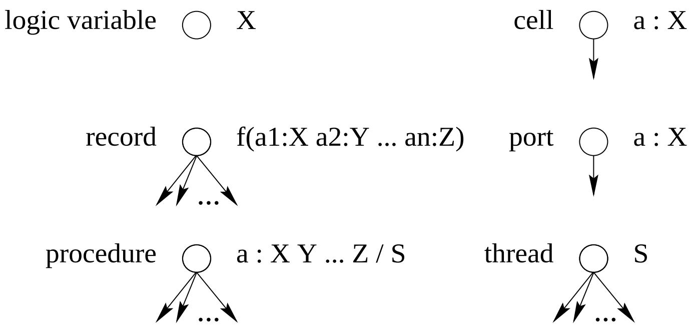

Fig. 7. Nodes in the language graph.

assume the exchange is invoked on site q and that the content-edge is on site p, where $1 \le p, q \le k$, then the distributed reduction rule is

$$\begin{array}{c||c} \texttt{\{Exchange C X Y\}}_q & (\texttt{X=Z})_q \\ \hline \sigma_r \ \wedge\ \texttt{C}_1\texttt{=n} \ \wedge \ldots \wedge\ \texttt{C}_k\texttt{=n} \ \wedge & \sigma_r \ \wedge\ \texttt{C}_1\texttt{=n} \ \wedge \ldots \wedge\ \texttt{C}_k\texttt{=n} \ \wedge \\ (\texttt{n:Z})_p \ \wedge\ 1 \le \forall i \le k, i \neq p : (\texttt{n:}\bot)_i & (\texttt{n:Y})_q \ \wedge\ 1 \le \forall i \le k, i \neq q : (\texttt{n:}\bot)_i \end{array}$$

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

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

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

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

6.2 The Graph Model

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

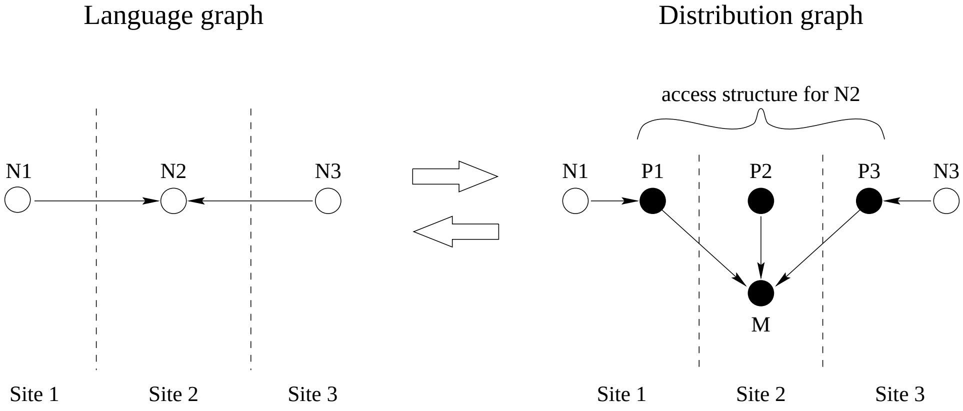

Fig. 8. From language graph to distribution graph.

6.2.1 *The Distribution Graph.* The distributed execution of OPM is introduced in two steps. In the first step, we model an OPM computation space as a graph, called *language graph.* Each Oz 2 entity except for an object corresponds to one node in the language graph (see Figure 7). An object is a derived concept that is modeled as a subgraph, namely a procedure with references to the object's state, lock, and methods. OPM execution is modeled as a sequence of graph transformations.

In the second step, we extend the language graph with the notion of *site.* First introduce a finite set of sites, and then annotate each node of the language graph with a site. If a node is referenced by a node on another site, then we map it to a *set* of nodes. This set is called the *access structure* of the original node (see Figure 8). An access structure consists of a set of *global nodes*, namely one *proxy node* per site and one *manager node* for the whole structure. In a cell access structure, the content-edge is an edge from exactly one proxy node to the content. Nodes that are only accessed locally (*local nodes*) do not have an access structure. In this case, the content-edge is an edge from the local node to the content.

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

6.2.2 *Mobility in the Distribution Graph.* At this point, it is useful to clarify how cell mobility fits into the distribution graph model. First, the nodes of the distribution graph never change sites. A manager node has a global address that is unique across the network and never changes. This makes memory management very simple, as explained in Section 7.2. Second, access structures can move across the network (albeit slowly) by creating proxies on fresh sites and by losing local references to existing proxies. Third, a content-edge can change sites (quickly) if requested to do so by a remote exchange. This is implemented by a change of state in the cell proxies that is coordinated by the mobile state protocol.

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

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

6.2.3 *The Failure Model.* The failure model must reliably inform the programmer if and when a failure occurs and allow him or her to take the necessary actions. The failure model is still under discussion, so the final design will likely differ from the one presented here. This section summarizes an extension to the mobile state protocol that provides precise failure detection. The programmer can enable a failure to appear in the language as an exception. Objects based on the extended protocol can be used as building blocks to program reliable objects. For example, one can program a reliable cell that has a primary-slave architecture [Coulouris et al. 1994].

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

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

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

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

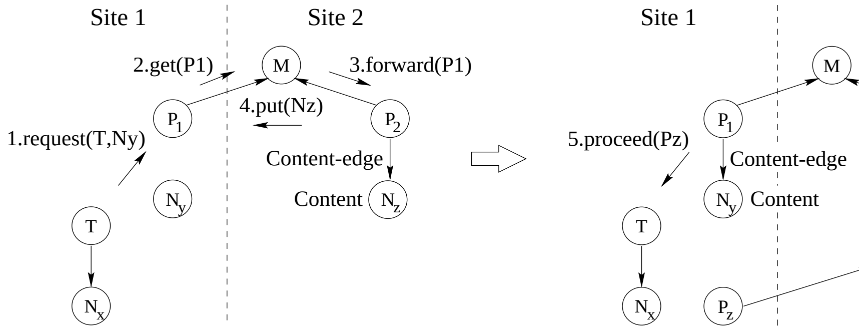

Fig. 9. Exchange initiates migration of content-edge.

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

6.3 Informal Description

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

The content-edge is stored at one of the cell proxies. Cell proxies exchange messages with threads in the engine. To ask for the cell content, a thread sends a message to a proxy. The thread then blocks waiting for a reply. After executing its protocol, the proxy sends a reply giving the content. This lets the thread do the binding. Figure 9 shows how this works. We assume that the content-edge is not at the current proxy. A proxy requests the content-edge by sending a message to the manager. The manager serializes possible multiple requests and sends forwarding commands to the proxies. The current location of the content-edge may lag behind the manager's knowledge of who is the eventual owner. This is all right: the content-edge will eventually be forwarded to every requesting site.

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

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

Outline of protocol (Figure 9).

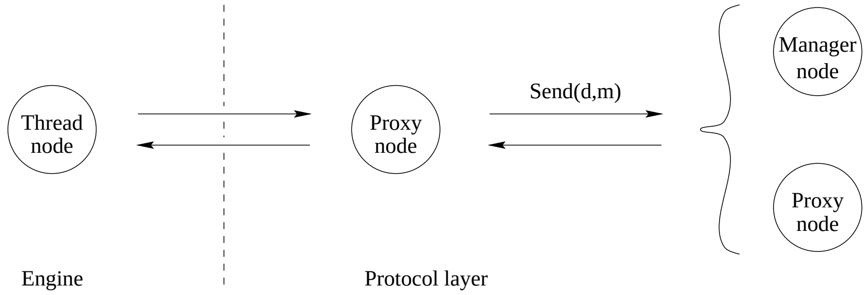

Fig. 10. Interface between engine and protocol.

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

(2) Manager M receives **get(P$_\mathbf{1}$)**. Manager sends **forward(P$_\mathbf{1}$)** to the current owner P_2 of the content-edge and updates the current owner to be P_1.

(3) Proxy P_2 receives **forward(P$_\mathbf{1}$)**. If P_2 has the content-edge, which points to N_z, then it sends **put(N$_\mathbf{z}$)** to P_1 and invalidates its content-edge. Otherwise, wait until the content-edge arrives at P_2. Sending the message **put(N$_\mathbf{z}$)** causes the creation of a new access structure for N_z.[15] From this point onward, all references to N_z are converted to P_z.

(4) Proxy P_1 receives **put(P$_\mathbf{z}$)**. At this point, the content-edge of P_1 points to N_y. P_1 then sends **proceed(P$_\mathbf{z}$)** to thread T.

(5) Thread T receives **proceed(P$_\mathbf{z}$)**. The thread then invokes the binding of N_x and P_z.

6.4 Formal Specification

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

6.4.1 *Preliminaries.* Consider a single manager node M, a set of k proxy nodes P_i with $1 \leq i \leq k$, and a set of m thread nodes T_i with $1 \leq i \leq m$. All nodes have state, can send messages to each other according to Figure 10, and can perform internal operations. Let these nodes be linked together by a network N that is a multiset containing messages of the form $d : m$ where d identifies a destination (proxy, manager, or thread node) and where m is a message.

The protocol is defined using reduction rules of the form

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

Table III. Node State

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

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

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

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

6.4.2 *Node State.* The node state is represented as a set of attributes of each node. Table III lists the attributes for proxy, manager, and thread nodes, along with their initial values. This table assumes without loss of generality that the content-edge is initially at proxy P_1. A *NodeRef* is a reference to any node. GetManagerRef() returns a reference to the manager M. GetProxyRef(i) returns a reference to proxy P_i. GetThreadRef(i) returns a reference to thread T_i. N is a *NodeRef* giving the initial content of the cell. NULL is a special value that marks an attribute as not valid. P.manager, P.id, and T.id are constants.

6.4.3 *Rule Definitions.* The protocol is defined in two parts. Figure 11 defines the procedure exchange_and_bind, which is part of the thread. Figure 12 defines the protocol rules. Rules 1–5 are defined for proxy nodes. Rule 6 is defined for the manager node.

The reduction of both the statement `{Exchange C X Y}` and its resulting binding `X=Z` is implemented in thread T by calling exchange_and_bind(P,N_X,N_Y). The nodes

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

Fig. 11. Mobile state protocol: Thread interface.

P, N_X, and N_Y correspond to the variables C, X, and Y. T sends a message containing N_Y to the proxy P on its site. T blocks until the content-edge and the content have moved to its site. Then the proxy sends the old content N_Z to T. The exchange is completed, strictly speaking, when N_Z is received in T, because that is when the bind operation becomes applicable. The thread then invokes bind(N_X,N_Z).

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

P.state=FREE if P has not requested the content-edge. P.state=CHAIN if P has requested the content-edge, which may not have arrived yet. P.content≠NULL if and only if P has the content-edge. P.forward≠NULL if and only if P should forward the content when it is present. P.thread and P.newcontent are used when the content-edge is remote. They store the information necessary for a correct reply when the content-edge arrives locally.

6.4.4 *Extension for Port Mobility.* The port protocol is an extension of the cell protocol defined in the previous section. As explained in Section 5.3, a port has two operations, send and localize, which are initiated by a thread referencing the port. The localize operation uses the same protocol as the cell exchange operation. For a correct implementation of the send operation, the port protocol must maintain the FIFO order of messages even during port movement. Furthermore, the protocol is defined so that there are no dependencies between proxies when moving a port. This means that a single very slow proxy cannot slow down a localize operation.

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

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

1. Request content (content not present).

$$\frac{\textbf{Receive}(\text{P,request(T,}N_y\text{))} \;\wedge\; \text{P.state=FREE}}{\begin{array}{l}\textbf{Send}(\text{P.manager,get(P.id))}\\ \text{P.state} \leftarrow \text{CHAIN}\\ \text{P.thread} \leftarrow \text{T}\\ \text{P.newcontent} \leftarrow N_y\end{array}}$$

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

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

3. Accept content and reply to thread.

$$\frac{\textbf{Receive}(\text{P,put(}N_z\text{))}}{\begin{array}{l}\text{P.content} \leftarrow N_z\\ \textbf{Send}(\text{P.thread,proceed(P.content))}\\ \text{P.content} \leftarrow \text{P.newcontent}\end{array}}$$

4. Accept forward.

$$\frac{\textbf{Receive}(\text{P,forward(P'))})}{\text{P.forward} \leftarrow \text{P'}}$$

5. Forward content.

$$\frac{\text{P.forward}\neq\text{NULL} \;\wedge\; \text{P.content}\neq\text{NULL}}{\begin{array}{l}\textbf{Send}(\text{P.forward,put(P.content))}\\ \text{P.forward} \leftarrow \text{NULL}\\ \text{P.content} \leftarrow \text{NULL}\\ \text{P.state} \leftarrow \text{FREE}\end{array}}$$

6. Serialize content requests (at manager).

$$\frac{\textbf{Receive}(\text{M,get(P))})}{\begin{array}{l}\textbf{Send}(\text{M.tail,forward(P))}\\ \text{M.tail} \leftarrow \text{P}\end{array}}$$

Fig. 12. Mobile state protocol: Migration of the content-edge.

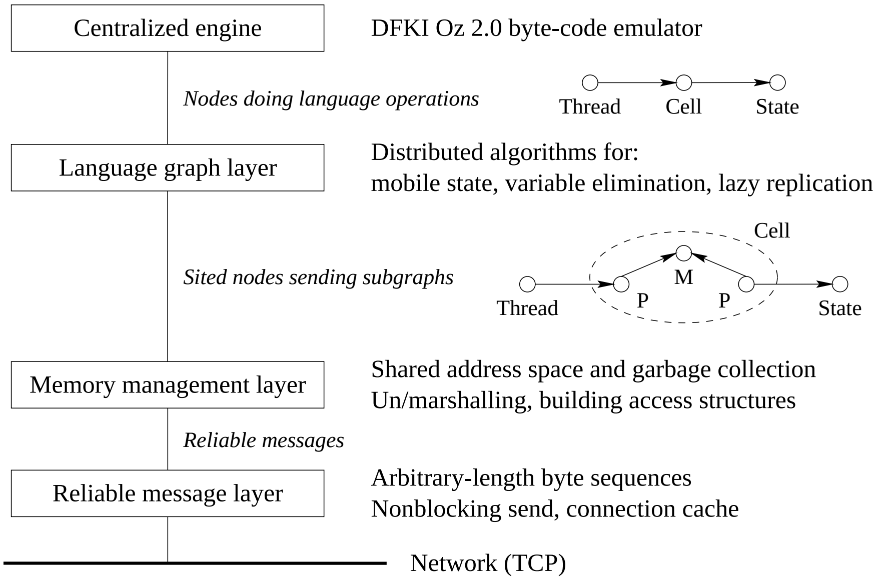
Centralized engine
DFKI Oz 2.0 byte-code emulator
Nodes doing language operations
Thread
Cell
State
Language graph layer
Distributed algorithms for:
mobile state, variable elimination, lazy replication
Sited nodes sending subgraphs
Cell
M
Thread
P
P
State
Memory management layer
Shared address space and garbage collection
Un/marshalling, building access structures
Reliable messages
Reliable message layer
Arbitrary-length byte sequences
Nonblocking send, connection cache
Network (TCP)

Fig. 13. System architecture on one site.

7. SYSTEM ARCHITECTURE

The mobile state protocol defines the distributed semantics of cells. This protocol is only one aspect of a Distributed Oz implementation. Other important aspects include the interface between the protocol and the centralized execution, how the protocol is built on top of a shared computation space, and how the shared computation space is built. This section summarizes these aspects in sufficient detail to situate the protocol. Figure 13 shows an architecture for the execution of the distribution graph. This architecture is fully implemented and is being used for application development. Sections 7.1, 7.2, and 7.3 summarize the language graph layer, the memory management layer, and the reliable message layer.

Distribution is added as a conservative extension to a *centralized engine*. The extension is designed not to affect the centralized performance. The centralized engine executes internally all operations on local nodes. It is based on emulator technology and has similar or better performance than current Java emulators [Henz 1997]. In particular, DFKI Oz 2.0 threads are much cheaper than Java threads. Operations on distributed entities are passed to the language graph layer. The *language graph layer* implements the distributed semantics for all Oz 2 entities, e.g., it decides when to do a local operation or a network communication. The language graph layer rests on a "bookkeeping" layer, the *memory management layer*. This layer implements the shared computation space, the building of access structures, and the distributed garbage collection. The *reliable message layer* implements transfer of arbitrary-length byte sequences between sites. It keeps a connection cache between sites and manages the message buffers. The *network* is the network interface of the host operating system, providing standard protocols such as TCP/IP.

Table IV. Object Granularity in the Distributed Oz Implementation

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

7.1 Language Graph Layer

The distribution semantics of each Oz 2 entity is implemented in the language graph layer. Each entity has a separate protocol. There are three essentially different protocols: mobile state (cells, objects, and ports), variable elimination (logic variables), and lazy replication (procedures). Records and threads have trivial protocols. As illustrated in Figure 13 for cells, these protocols implement the illusion that all entities are centralized.

Table IV gives the space usage of objects and messages in the mobile state protocol. Objects have a tag that defines them to be local, active proxy, passive proxy, or manager. This tag is combined with other fields to take up no extra space. The following extra run-time overhead is paid over a system that has only local objects. For each object operation, test the tag to see if the object is local or global, and if it is global, check if the content-edge is local. The local objects shown are of minimum size; add 4 bytes for each attribute and method. The global object sizes include the overhead for distributed garbage collection (see Section 7.2.3). A *passive* proxy is one that has not been called since the most recent local garbage collection. Other proxies are *active*. In the **put** message, S denotes the marshalled size of the state.

7.2 Memory Management Layer

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

7.2.2 *Building Access Structures.* Access structures are built and managed automatically when language entities become remotely referenced. This happens whenever messages exchanged between nodes on different sites contain references to other nodes. If the reference is to a local node, then the memory management layer converts the local node into an access structure. We say the local node is *globalized* (see Figure 14). While the message is in the network, the access structure consists of a manager and one proxy. When the message arrives at the destination site, then

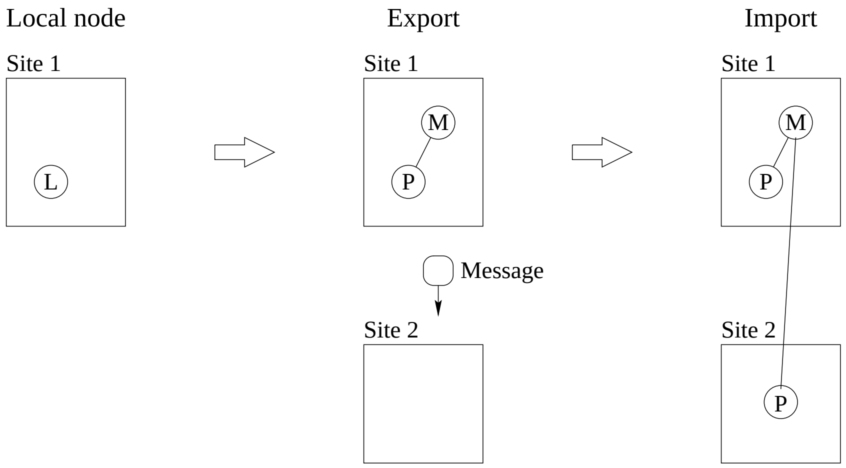

Fig. 14. Globalizing a local node.

a new proxy is created there. Access structures can reduce in size and disappear completely through garbage collection.

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

A global address is reclaimed when the node that it refers to is no longer remotely referenced. This is done by the credit mechanism, which is informed by the local garbage collectors. This scheme recovers all garbage except for cross-site cycles. The only cross-site cycles in our system occur between different objects or cells. Since records and procedures are both replicated, cycles between them will be localized to single sites. The credit mechanism does not suffer from the memory or network inefficiencies of previous reference-counting schemes [Plainfossé and Shapiro 1995].

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

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

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

7.3 Reliable Message Layer

The reliable message layer is the part of the Distributed Oz architecture that is closest to the operating system. This layer assumes a reliable transport protocol with no bounds on transfer time. For all entities except threads and ports, no assumptions are made on relative ordering of messages (no FIFO). For threads and ports, FIFO connections are made. The current prototype implements a cache of TCP connections to provide reliable transfer between arbitrary sites on a wide-area network [Comer 1995]. Recent implementations of TCP can outperform UDP [Callaghan 1996]. To send arbitrary-length messages from fair concurrent threads, the implementation manages its own buffers and uses nonblocking send and receive system calls. If some global addresses in a message require additional credit, then the message is put in a pending queue until all credit arrives.

8. RELATED WORK

All systems that we know of except Emerald [Jul et al. 1988] and Obliq [Cardelli 1995] do distributed execution by adding a distribution layer on top of a centralized language, e.g., CORBA [Crowcroft 1996; Otte et al. 1996], DCE [Tanenbaum 1995], Erlang [Wikström 1994], Java [Sun Microsystems 1996], Facile [Leth and Thomsen 1992], and Telescript [General Magic]. 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. In the following sections, we take a closer look at distributed shared memory, Emerald, and Obliq.

A better environment for distributed programming can be obtained by looking carefully at the entities of the language and conservatively extending their behavior to a distributed setting. For example, the Remote Procedure Call (RPC) [Birrell and Nelson 1984] is designed to mimic centralized procedure calling and is therefore a precursor to the design given in this article. Following this integrated approach has two consequences. First, in order to carry it out successfully, the language must have a well-defined operational semantics that clearly identifies the entities. Second, to do the extensions right one must design a distributed algorithm for each entity.

8.1 Distributed Shared Memory

Distributed shared memory (DSM) [Coulouris et al. 1994; Tanenbaum 1995] has the potential to provide an adequate substrate for distributed programming. DSM has traditionally been viewed as a substrate for parallel programming, but work has been done on using it for distributed programming. We limit the discussion to distribution in software DSM. To achieve predictable network awareness, a language for distributed programming must be well-matched with its DSM layer. Following Coulouris et al. [1994], we distinguish between page-based and library-based DSM.

Page-based DSM does not provide predictable network awareness. The units of distribution ("pages") do not correspond directly to language entities. This is too coarse grained for the applications we have in mind. It leads to false sharing. Munin [Carter et al. 1991], while page-based, provides programmer annotations for network awareness. A data item in memory can be annotated as read-only, migratory, write-shared, and so forth.

Library-based DSM provides sharing that is designed for particular data abstractions and hence can avoid the problems of granularity and false sharing. For example, Orca [Bal et al. 1992] provides network pointers, called "general graphs," and shared objects. Orca is designed for parallel applications. Linda [Carriero and Gelernter 1992] provides operations to insert and extract immutable data items, called "tuples," from a shared space. This is called a *coordination model*, and it can be added to any existing language. Linda does not address the language issues of network transparency.

Distributed Oz follows the library-based approach. The shared computation space is a DSM layer that is designed to support all language entities. The layer is extended to provide functionality that is not part of traditional DSMs. First, it supports single-assignment data in a strong form (logic variables) as well as other sharing protocols such as read-only data (values) and migratory data (objects). Second, as was briefly mentioned in Section 2, the system is open: sites can connect and disconnect dynamically. Although not impossible, we do not know of any DSM system that possesses this property. Third, the system is portable across a wide range of operating systems and processors. Fourth, the system can be extended to support precise failure detection.

8.2 Emerald

Emerald is a statically typed concurrent object-based language that provides fine-grained object mobility [Jul 1988; Jul et al. 1988]. The object system of Emerald is interesting in its own right. We limit the discussion to issues related to distribution. Emerald has distributed lexical scoping and is implemented efficiently with a two-level addressing scheme. Emerald is not an open system. Objects can be mutable or immutable. Objects are stationary by default and explicit primitive operations exist to move them. Having an object reference gives both the right to call and to move the object; these rights are separated in Distributed Oz. Immutable objects are copied when moved. Apart from object mobility, Emerald does not provide any special support for latency tolerance. There is no syntactic support for using objects as caches.

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

Because of the aliasing chain and possible broadcasting, it is difficult or impossible to predict the network behavior in Emerald or to guarantee that an object is independent of third-party sites. These problems are solved in Distributed Oz by using a manager node that is known to all proxies (see Section 6.2.2). This gives an upper bound of three on the number of network hops to get the object and guarantees that all third-party dependencies except for the manager site eventually disappear. Furthermore, the lack of an aliasing chain means that losing an object is so infrequent that it is considered as an object failure. There is therefore no need for a broadcast algorithm.

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

8.3 Obliq

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

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

Distributed Oz takes this approach for the complete language, consisting of seven language entities. Each of these entities has a distributed algorithm that is used to remotely perform an operation on the entity (see Figure 3). The algorithms are designed to preserve the language semantics while providing a simple model for the communication patterns.

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

Oz 2 objects are defined as procedures that have access to a cell. The content-edge refers to the current object state. Mobility is obtained by making the cell mobile. When a method is invoked on a remote site, the content-edge is first moved to that site. Integrity of the state is preserved because the cell continues to obey the language semantics. The implementation uses a distributed algorithm to move the content-edge (see Section 6). Because mobility is part of a cell's distributed semantics, there are no chains of indirections. This is true as well for mostly stationary objects, which use the port protocol (see Section 6.4.4).

9. CONCLUSIONS, STATUS, AND CURRENT WORK

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

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

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

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

We outline the system architecture including the distributed memory management and garbage collection. A prototype implementation of Distributed Oz exists that incorporates all of the ideas presented in this article. The prototype maintains the semantics of the Oz 2 language. The implementation is an extension of the centralized DFKI Oz 2.0 system and has been developed jointly by the German Research Center for Artificial Intelligence (DFKI) [Smolka et al. 1995] and the Swedish Institute of Computer Science (SICS). DFKI Oz 2.0 is publicly available[16]

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

and has a full-featured development environment.

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

APPENDIX

A. CORRECTNESS PROOF OF MOBILE STATE PROTOCOL

This appendix gives a proof that the mobile state protocol as defined in Section 6.4 implements the language semantics of the exchange operation as defined in Section 6.1. We have given three specifications of the exchange operation:

(LS)	Language reduction rule (Section 6.1)
(DS)	Distributed reduction rule (Section 6.1)
(MP)	Mobile state protocol (Section 6.4)

Theorem T1 (Section 6.1) implies that (DS) implements (LS). It remains to be shown that (MP) implements (DS). We do this in two parts. First, in Section A.1 we prove safety and liveness of the mobile state protocol. Namely, all reachable configurations satisfy the *chain invariant* defined in Section A.1.1, and a request for the content-edge on a node that does not have it causes it to arrive exactly once. Then, in Section A.2 we use these results first to prove that the mobile state protocol is observationally equivalent to a much simpler protocol. We then show that the latter implements the distributed reduction rule for exchange.

A.1 Mobile State Protocol Correctly Migrates the Content-edge

This section proves that the mobile state protocol correctly implements the migration of the content-edge. We follow the definitions and notations introduced in Section 6. The proof is structured around a global distributed data structure that we call a *chain*, which is defined in Section A.1.1 by an invariant of the distribution graph [Lynch 1996; Tel 1994]. Section A.1.2 proves that the mobile state protocol satisfies the chain invariant. Section A.1.3 proves that requesting the content causes it to arrive exactly once.

Informally, a chain consists, at any given instant, of the known path of the content among the proxy nodes (see Figure 15). That is, it is a sequence of proxy nodes such that the first node contains the content or will eventually receive it and that all nodes will eventually pass the content on to the next node in the chain. The chain grows if new content-edge requests are more frequent than the rate at which the content is forwarded among proxies. If there are no new requests, then the chain shrinks to length one. The chain is defined formally as part of the proof. Reasoning in terms of the chain makes proving properties of the protocol tractable.

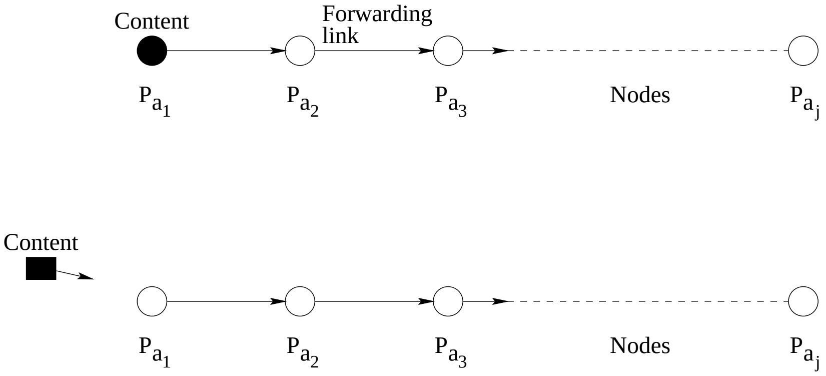

Fig. 15. Two forms of a chain.

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

A.1.1 *Chain Invariant.* The chain invariant I is defined as follows on the distribution graph:

$$I = I_p \wedge I_a \wedge I_b \wedge I_c \wedge I_t \wedge I_u \tag{1}$$

$$I_p = A, B, C \text{ form a partition of } \{1, ..., k\} \tag{2}$$

$$\begin{aligned} I_a = {} & A = \{a_1, ..., a_j\} \wedge j > 0 \wedge \\ & \left(1 \le \forall i < j : \begin{cases} P_{a_i}.\text{state} = \text{CHAIN} \\ P_{a_i}.\text{forward} \in \{\text{NULL}, a_{i+1}\} \\ P_{a_i}.\text{forward} = a_{i+1} \;\oplus\; a_i : \text{forward}(a_{i+1}) \in N \end{cases} \right) \wedge \\ & P_{a_j}.\text{state} = \text{CHAIN} \wedge P_{a_j}.\text{forward} = \text{NULL} \wedge M.\text{tail} = a_j \end{aligned} \tag{3}$$

$$I_b = \forall i \in B : \begin{pmatrix} P_i.\text{state} = \text{CHAIN} \wedge P_i.\text{forward} = \text{NULL} \wedge \\ M : \text{get}(i) \in N \end{pmatrix} \tag{4}$$

$$I_c = \forall i \in C : (P_i.\text{state} = \text{FREE} \wedge P_i.\text{forward} = \text{NULL}) \tag{5}$$

$$\begin{aligned} I_t = {} & P_{a_1}.\text{content} \in \{\text{NULL}, \text{CVAL}\} \wedge \\ & (P_{a_1}.\text{content} = \text{CVAL} \;\oplus\; a_1 : \text{put}(\text{CVAL}) \in N) \wedge \\ & 1 \le \forall i \le k,\ i \ne a_1 : P_i.\text{content} = \text{NULL} \end{aligned} \tag{6}$$

$$I_u = \text{All messages in } N \text{ are unique and explicitly mentioned in } I \tag{7}$$

Informally, I_a states that all the proxy nodes in A form a chain, where $\oplus$ denotes the exclusive-or (see Figure 15). We call P_{a_1} the *head* of the chain and P_{a_j} the *tail* of the chain. I_b states that all the proxy nodes in B will eventually become part of the chain. When it is known from which source node a target node will receive

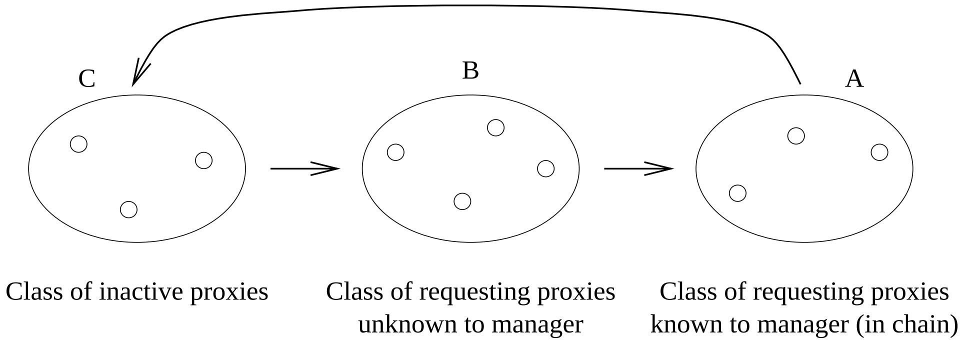
C
B
A
Class of inactive proxies
Class of requesting proxies unknown to manager
Class of requesting proxies known to manager (in chain)

Fig. 16. Classifying the proxy nodes.

the content, then the target node is considered part of the chain. I_c states that all other proxy nodes (which are in C) are not part of the chain. The content invariant I_t states that there is exactly one content-edge in the system and that it belongs to the head of the chain. Writing the uniqueness invariant I_u as an explicit formula is left as an exercise for the reader.

Figure 16 illustrates the partitioning of the proxy nodes in classes C, B, and A. During execution, a proxy node changes class according to the arrows. The node is in class C when it does not have the content-edge and has not requested it. The node moves from class C to B when the content-edge is requested. The node moves from class B to A when the manager is informed of the request. The node moves from class A to C when the content-edge leaves the node. Within class A, the content-edge moves from one node to the next in the chain. The node reaches the head of the chain when it receives the content-edge.

A.1.2 *Safety Theorem*

Theorem S. *The chain invariant (formula I in Section A.1.1) is an invariant of the mobile state protocol (Section 6.4).*

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

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

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

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

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

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

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

This proves the theorem. □

A.1.3 *Liveness Theorem*

THEOREM L. *Given the fairness assumptions of Section 6.4.1, requesting the content at a proxy node will cause it eventually to arrive once.*

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

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

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

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

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

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

$$A_{x+1} = A_x \quad (1)$$

$$A_{x+1} = A_x \setminus \{a_1\} \quad (2)$$

$$A_{x+1} = A_x \cup \{a_{j+1}\} \quad (3)$$

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

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

A.2 Mobile State Protocol Implements Distributed Semantics

The proof is presented as two lemmas and a theorem. From the two lemmas it follows that the mobile state protocol is observationally equivalent to a much-simplified protocol with one proxy node, one reduction rule, and no manager. In a similar manner to Section 6.1.3, we then use abstraction to show that the behavior of this simplified protocol exactly corresponds to the distributed semantics of exchange.

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

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

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

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

(2) P.state=FREE ∧ P.content=NULL. Rule 1 is applicable and stays applicable, so by the fairness assumption eventually it is reduced.

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

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

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

(1) Rule 1 is reduced. By Theorem L, the content arrives exactly once by the reduction of rule 3. This makes true the second condition P.content≠NULL of the (EXCH) rule. Between the reduction of rule 1 and the arrival of the content, only rule 4 is potentially applicable. Reducing rule 4 is irrelevant since it only changes the value of P.forward, which does not change the value of any other

Table V. Correspondence between distributed semantics and graph notation

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

node attribute nor the applicability of any rules. When the content arrives, rule 3 becomes applicable. Reduction of rule 3 makes the lemma true.

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

This proves the lemma. □

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

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

Theorem T2. *The mobile state protocol of Section 6.4 implements the distributed semantics of Section 6.1.*

Proof. We first make clear the correspondence between the notations of Sections 6.1 and 6.4 (Table V). Then we show that the execution of the exchange procedure follows exactly the distributed reduction rule.

By Lemma A, executing exchange_and_bind(P,$\mathrm{N_X}$,$\mathrm{N_Y}$) in thread T on site q eventually reduces one (EXCH) rule. The bind operation becomes applicable when the message sent by this rule is received by the thread. The receive therefore marks the end of the exchange reduction. Denote P.content=$\mathrm{N_Z}$ just before entering the (EXCH) rule's body. By Lemma B, $\mathrm{N_Z}$ is the result of the previous exchange. Then, assuming the correspondence in Table V holds just before the (EXCH) rule, we show that the correspondence holds just after the (EXCH) rule:

(1) The nodes of the access structure are undisturbed.

(2) P_q.content=$\mathrm{N_Y}$ and by Theorem S, $1 \leq \forall i \leq k, i \neq q$: P_i.content=NULL.

(3) The operation bind($\mathrm{N_X}$,$\mathrm{N_Z}$) is applicable in thread T after the exchange.

This corresponds exactly to the distributed reduction rule of Section 6.1. □

ACKNOWLEDGEMENTS

We thank Luc Onana for fruitful discussions that led to mobile ports and to the proof given in Appendix A.1. We thank Christian Schulte for his help with the shared editor. We thank Michel Sintzoff, Joachim Niehren, and the anonymous referees for their perceptive comments that permitted us to improve and complete the presentation.

REFERENCES

Abelson, H., Sussman, G. J., and Sussman, J. 1985. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass.

Armstrong, J., Williams, M., Wikström, C., and Virding, R. 1996. *Concurrent Programming in Erlang.* Prentice-Hall, Englewood Cliffs, N.J.

Axling, T., Haridi, S., and Fahlen, L. 1995. Concurrent constraint programming virtual reality applications. In the *2nd International Conference on Military Applications of Synthetic Environments and Virtual Reality* (*MASEVR 95*). Defence Material Administration, Stockholm, Sweden.

Bal, H. E., Kaashoek, F. E., and Tanenbaum, A. S. 1992. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng. 18,* 3 (Mar.), 190–205.

Bal, H. E., Steiner, J. G., and Tanenbaum, A. S. 1989. Programming languages for distributed computing systems. *ACM Comput. Surv. 21,* 3 (Sept.), 261–322.

Barth, P. S., Nikhil, R. S., and Arvind. 1991. M-structures: Extending a parallel, nonstrict, functional language with state. In *Functional Programming and Computer Architecture.* Springer-Verlag, Berlin.

Birrell, A. D. and Nelson, B. J. 1984. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2,* 1 (Feb.), 39–59.

Callaghan, B. 1996. WebNFS—The file system for the World-Wide Web. White paper, Sun Microsystems, Mountain View, Calif. May.

Cardelli, L. 1995. A language with distributed scope. *ACM Trans. Comput. Syst. 8,* 1 (Jan.), 27–59. Also appeared in POPL 95.

Carlsson, C. and Hagsand, O. 1996. DIVE—A platform for multi-user virtual environments. *Computers and Graphics 17,* 6.

Carriero, N. and Gelernter, D. 1992. Coordination languages and their significance. *Commun. ACM 35,* 2 (Feb.), 96–107.

Carter, J. B., Bennett, J. K., and Zwaenepoel, W. 1991. Implementation and performance of Munin. In the *13th ACM Symposium on Operating System Principles.* ACM, New York, 152–164.

Comer, D. E. 1995. *Internetworking with TCP/IP. Vol. 1: Principles, Protocols, and Architecture.* Prentice-Hall, Englewood Cliffs, N.J.

Coulouris, G., Dollimore, J., and Kindberg, T. 1994. *Distributed Systems Concepts and Design 2nd ed.* Addison-Wesley, Reading, Mass.

Crowcroft, J. 1996. *Open Distributed Systems.* University College London Press, London, U.K.

Deering, S. 1989. Host extensions for IP multicasting. Tech. Rep. RFC1112. Aug.

Ericsson. 1996. *Open Telecom Platform—User's Guide, Reference Manual, Installation Guide, OS Specific Parts.* Telefonaktiebolaget LM Ericsson, Stockholm, Sweden.

Fischer, K., Kuhn, N., and Müller, J. P. 1994. Distributed, knowledge-based, reactive scheduling in the transportation domain. In the *10th IEEE Conference on Artificial Intelligence and Applications.* IEEE, New York.

Fischer, K., Muller, J. P., and Pischel, M. 1995. A model for cooperative transportation scheduling. In the *1st International Conference on Multiagent Systems* (*ICMAS 95*). 109–116.

Foody, M. 1997. Let's talk (Special report building networked applications). *BYTE 22,* 4 (Apr.), 99–102.

Halstead, R. H. 1985. MultiLisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7,* 4 (Oct.), 501–538.

Haridi, S. 1996. An Oz 2.0 tutorial. Available at `http://sics.se/~seif/oz.html`.

Haridi, S., Van Roy, P., and Smolka, G. 1997. An overview of the design of Distributed Oz. In the *2nd International Symposium on Parallel Symbolic Computation* (*PASCO 97*). ACM, New York.

Henz, M. 1997. Objects in Oz. Doctoral dissertation, Univ. des Saarlandes, Saarbrücken, Germany.

HENZ, M., LAUER, S., AND ZIMMERMANN, D. 1996. COMPOzE—Intention-based music composition through constraint programming. In the *International Conference on Tools with Artificial Intelligence.* IEEE, New York.

HENZ, M. AND WÜRTZ, J. 1996. Using Oz for college timetabling. In the *International Conference on the Practice and Theory of Automated Timetabling*, E. K. Burke and P. Ross, Eds. Lecture Notes in Computer Science, vol. 1153. Springer-Verlag, Berlin, 162–177.

IANNUCCI, R. A. 1990. *Parallel Machines: Parallel Machine Languages. The Emergence of Hybrid Dataflow Computer Architectures.* Kluwer, Dordrecht, the Netherlands.

JUL, E. 1988. Object mobility in a distributed object-oriented system. Ph.D. thesis, Univ. of Washington, Seattle, Wash.

JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. 1988. Fine-grained mobility in the Emerald system. *ACM Trans. Comput. Syst. 6,* 1 (Feb.), 109–133.

LAMPSON, B. W. 1993. Reliable messages and connection establishment. In *Distributed Systems*, S. Mullender, Ed. Addison-Wesley, Reading, Mass., 251–281.

LETH, L. AND THOMSEN, B. 1992. Some Facile chemistry. Tech. Rep. ECRC-92-14, ECRC, Munich, Germany. May.

LYNCH, N. 1996. *Distributed Algorithms.* Morgan Kaufmann, San Francisco, Calif.

GENERAL MAGIC. *Telescript Developer Resources.* General Magic, See `http://www.genmagic.com/`.

SUN MICROSYSTEMS. 1996. *The Java Series.* Sun Microsystems, Mountain View, Calif. Available at `http://www.aw.com/cp/javaseries.html`.

OTTE, R., PATRICK, P., AND ROY, M. 1996. *Understanding CORBA: The Common Object Request Broker Architecture.* Prentice-Hall PTR, Upper Saddle River, N.J.

PLAINFOSSÉ, D. AND SHAPIRO, M. 1995. A survey of distributed garbage collection techniques. In the *International Workshop on Memory Management.* Lecture Notes in Computer Science, vol. 986. Springer-Verlag, Berlin, 211–249.

SCHMEIER, S. AND ACHIM, S. 1996. PASHA II—Personal assistant for scheduling appointments. In the *1st International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology* (*PAAM 96*). The Practical Application Company, Lancashire, United Kingdom.

SHAPIRO, E. 1989. The family of concurrent logic programming languages. *ACM Comput. Surv. 21,* 3 (Sept.), 413–510.

SMOLKA, G. 1995a. *An Oz Primer.* Programming Systems Lab, Univ. des Saarlandes, Saarbrücken, Germany. Available at `http://www.ps.uni-sb.de`.

SMOLKA, G. 1995b. The Oz programming model. In *Computer Science Today.* Lecture Notes in Computer Science, vol. 1000. Springer-Verlag, Berlin, 324–343.

SMOLKA, G., HENZ, M., AND WÜRTZ, J. 1995. Object-oriented concurrent constraint programming in Oz. In *Principles and Practice of Constraint Programming*, P. Van Hentenryck and V. Saraswat, Eds. MIT Press, Cambridge, Mass., 29–48.

SMOLKA, G., SCHULTE, C., AND VAN ROY, P. 1995. PERDIO—Persistent and distributed programming in Oz. BMBF project proposal. Available at `http://www.ps.uni-sb.de`.

TANENBAUM, A. S. 1995. *Distributed Operating Systems.* Prentice-Hall, Englewood Cliffs, N.J.

TEL, G. 1994. *An Introduction to Distributed Algorithms.* Cambridge University Press, Cambridge, United Kingdom.

WALSER, J. P. 1996. Feasible cellular frequency assignment using constraint programming abstractions. In the *1st Workshop on Constraint Programming Applications, CP 96.*

WIKSTRÖM, C. 1994. Distributed programming in Erlang. In the *1st International Symposium on Parallel Symbolic Computation* (*PASCO 94*). World Scientific, Singapore, 412–421.

Received January 1997; revised April 1997; accepted May 1997