# ON THE SEPARATION OF CONCERNS IN DISTRIBUTED PROGRAMMING: APPLICATION TO DISTRIBUTION STRUCTURE AND FAULT TOLERANCE IN MOZART

PETER VAN ROY
*Université catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium*
*E-mail: pvr@info.ucl.ac.be*

Writing distributed applications is difficult because the programmer has to explicitly juggle many quite different concerns, including application functionality, distribution structure, fault tolerance, security, open computing, and others. An important goal is to separate the application functionality from the other concerns. This article presents one step towards that goal. We show how to integrate mutable pointers into a design that separates functionality, distribution structure, and fault tolerance. Mutable pointers, as a realization of explicit state, are an important data type that forms the basis for object-oriented programming. We start by defining mutable pointers in a centralized fault-free system. We then refine this definition by successively adding a distribution model and a failure model. The resulting semantics can be implemented efficiently and is a sufficient base to build nontrivial abstractions for fault tolerance. The design presented here is fully implemented as part of the Mozart Programming System (see `http://www.mozart-oz.org`).

## 1 Introduction

There are two basic ways to build a platform for distributed application development. Either one starts with an existing language, and adds functionality in terms of library functions, or one designs the language in tandem with the platform. The first approach is indicated for commercial developments, to maintain upward compatibility with existing investments. This approach is taken by CORBA [1] and Java [2]. The second approach is indicated for research purposes, to investigate what is appropriate in a language for distributed programming. This article gives an example of the second approach.

Our goal is to build a platform in which it is possible to separate as much as possible the different concerns of distributed computing. What are these concerns? In our view, they are consequences of the properties of a distributed system. We define a distributed system as a set of sites interconnected by an asynchronous network. In this article, we are interested in collaborative applications running on the Internet. We define a site as an operating system process and the network as the Internet. Characteristic properties of a distributed system include the lack of global information and global time, large and unpredictable communication delays between sites, and the possibility of

partial failures [3]. From this point of view, a site is a maximal subset of a distributed system that does not have these three properties. We assume that the system can be partitioned into a finite number of sites. We assume that the network is a fully connected graph between sites, i.e., we do not consider routing explicitly.

Four important concerns of a distributed application are (see [4]):

**Application functionality** What the application does if all effects of distribution are disregarded. This is the primary concern; it defines the application as if it were running on a single site.

**Distribution structure** The partitioning of the application over a set of sites.

**Fault tolerance** The ability for the application to continue providing its service despite partial failures in the implementation.

**Security** The ability for the application to continue providing its service despite intentional interference.

This article investigates the separation of the first *three* concerns, namely application functionality, distribution structure, and fault tolerance. We do this in the context of *mutable pointers*. A mutable pointer is one of the simplest ways to implement the notion of *state*, which is at the heart of object-oriented programming. Mutable pointers are not easy to implement well in a distributed system, which is why we choose them to illustrate our approach.

We use the following terminology, which is taken from [5]. With regard to a given layer in the implementation, a *fault* occurs when one of the layer's input operations deviates from its correct behavior. A *failure* occurs when the layer can no longer deliver the correct behavior to a higher layer. The Mozart implementation has three layers: distribution layer, network layer, and underlying operating system including network. A failure in the network shows up as a fault in the network layer. This may show up as a fault in the distribution layer, which in turn may result in failure of a language entity to perform correctly.

A system that completely separates functionality from distribution structure is called *network transparent* [6]. This means that a given program will run with exactly the same semantics independent of its distribution structure. To be precise, we assume an interleaving semantics that does not require knowing the time, but is based only on causal dependencies between instructions and a weak fairness condition. Section 3 gives a formal definition.

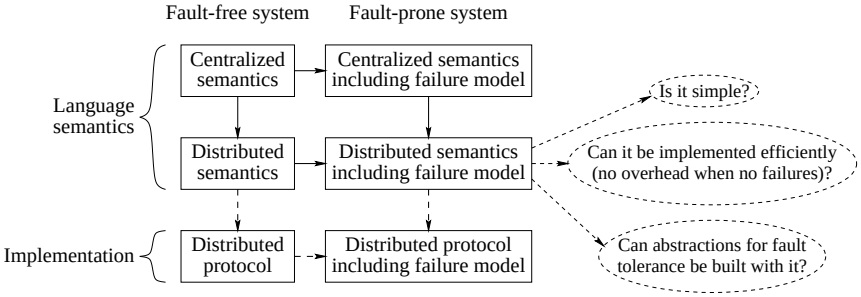



Figure 1. Refining the language semantics with distribution and fault tolerance

There are few systems that separate more than two concerns. Argus [7] and FT SR [8] both address distribution and fault tolerance. Each system implements an eponymous language that provides high-level fault tolerance abstractions. The FRIENDS project [9] has built a prototype system that partially separates the concerns of functionality, fault tolerance, security, and group communication. Our approach is complementary to these three projects. We focus on the semantics of language primitives. We ask three questions. How should fault tolerance show up in the language primitives? Can the primitives be implemented efficiently? How can one build fault tolerance abstractions within the language by using these primitives?

Separation of concerns is useful in a much wider context than just distributed programming. For example, lazy functional languages allow the separation of *what* is being calculated from *how much* of it is needed [10].

Section 2 gives an overview of our approach: the basic design principles and the semantic refinements. Section 3 briefly defines our execution model and the formalism we use to define the semantics. Section 4 gives the semantics of mutable pointers for centralized and distributed systems without faults, and outlines a distributed protocol that implements these semantics. Section 5 defines and justifies our failure model and gives the semantics of mutable pointers for distributed systems obeying this failure model. With this semantics, two questions remain: can the semantics be implemented efficiently, and does it suffice to build the abstractions we need for fault tolerance? Section 6 gives the properties of a distributed protocol that implements the fault tolerant distributed semantics. Section 7 lists the nontrivial abstractions for fault tolerance that we have built. Finally, Section 8 summarizes the main lessons learned and outlines what remains to be done.

## 2 The Design Approach

We first define the semantics of mutable pointers in a standard way for a centralized, fault-free language. We then refine this semantics in two steps to incorporate two additional concerns: distribution structure and fault tolerance (see Figure 1). These refinements follow two important design principles of the Mozart project:

1. Each layer of the system should contain the right primitives. This property is impossible to achieve *a priori*. To achieve it, we initially put only quite simple primitives in the lower layers, but we make them flexible enough to allow building more powerful abstractions at higher layers. As we learn which abstractions are useful and how to implement them, we modify the lower-layer primitives to be the *right* ones, i.e., to be simple and to allow efficient implementation of the abstractions we need.

2. The system should have a simple formal semantics. New primitive operations are defined carefully before being put in the system.

For fault tolerance, following these principles implies that the distributed semantics including failure model should be simple, implementable without overhead when there are no faults, and sufficiently flexible to allow nontrivial fault tolerance abstractions to be built. Sections 2.1 and 2.2 summarize how to apply this approach for fault-free and fault-prone systems. Sections 4 and 5 define the semantics and discuss some ancillary topics.

### *2.1 Fault-Free Systems*

A fault-free system is one whose components always work according to their specification. We consider the following three levels of refinement for fault-free systems:

- In Figure 1, we start at top left, with the centralized semantics of the mutable pointer and of its atomic read-and-write operation, which is called "exchange" (see Section 4.1). Exchange is quite a simple operation; it is defined with a single reduction rule.

- Following the arrow downwards, we refine the semantics to get the distributed semantics (see Section 4.2). In a distributed setting, the mutable pointer is at each instant located on a single site. The distributed semantics shows this, and also shows when and to which site the mutable pointer moves.

- Following the arrow downwards again, we refine the distributed semantics to get the distributed protocol that implements this semantics in terms of a message-passing architecture (see Section 4.3). The distributed protocol implements the mobility of the mutable pointer.

### *2.2 Fault-Prone Systems*

The refinements of the preceding section are valid when there are no faults. If the system components can have faults, i.e., the system is *fault-prone*, then the situation becomes more complex. To understand it, we first define a *failure model*, which enumerates what kind of failures our system is set up to detect and tolerate. The effect of the failure model shows up in the semantics at *all* levels; it is not at all like the distribution structure, which is completely hidden at the centralized level.

Why is it important that faults show up at the language level? Why is it wrong to define a fault tolerant algorithm with enough redundancy, so that the centralized semantics remains the same? The reason is simple: we did not want to hardwire a complex fault tolerant algorithm into the implementation. This is contrary to the first design principle. We prefer a simple failure detection ability in the distribution layer that allows us to build all the fault tolerance we need at the language level. Later, when we understand what the right primitives are, we migrate them to the implementation (the distribution and network layers) to regain any lost efficiency. Therefore, our distribution layer has a relatively simple protocol that satisfies two goals:

- There is no performance penalty if there is no fault (see Section 6).
- There are sufficient hooks to implement fault tolerance abstractions in the language (see Section 7). In fact, these "hooks" are just that part of the failure model that shows up in the language semantics.

It is not obvious that one can design such a protocol. In what follows, we progressively show that it can be done. First we give the semantics of mutable pointers in three steps, ending up with a simple semantics for fault-prone distributed systems (see Sections 3, 4, and 5). Along the way, we define the network-layer failure model (Section 5.1) and how it shows up for mutable pointers at the language level (Section 5.2). Then we show that this semantics can be implemented efficiently (see Section 6). Finally, we show that this semantics suffices to build interesting nontrivial abstractions for fault tolerance (see Section 7). This article does not give all the details of the implemented protocol nor of the fault tolerance abstractions. For more information, please see [11,12].

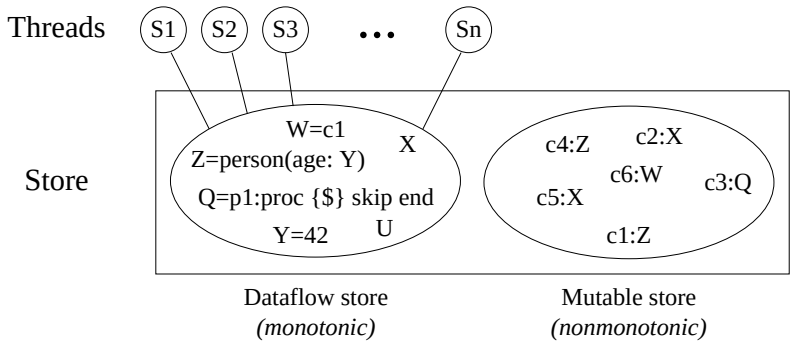



Figure 2. The Oz execution model

## 3 Notation and Concepts

Mutable pointers are part of the Oz language. This section defines the Oz execution model and a formalism for its semantics. Sections 4 and 5 define the semantics of mutable pointers. We only give the semantics of a single operation, namely the atomic read-and-write, which we call *exchange*. This allows us to give the essential insights with a minimum of notation. Adding the other operations is an easy exercise. The full Oz execution model also has support for logic programming and constraint programming [13,14]. We do not present this support here since it is outside the scope of the article.

### *3.1 Execution Model*

The Oz execution model consists of threads observing a shared store (see Figure 2) [15]. The store consists of two parts:

- A dataflow store, consisting of dataflow variables that are either unbound or bound. A bound variable references a term (i.e., atom, record, procedure, or name) whose arguments themselves may be bound or unbound.[a] Unbound variables can be bound to unbound variables, in which case they become identical references. The dataflow store is *monotonic*, i.e., bindings can only be added, not removed or changed.

- A mutable store, consisting of mutable pointers that contain references

[a] Technically, dataflow variables are a kind of single-assignment variable called *logic variable*; binding is done by distributed unification [16].

into the dataflow store.[b] (We use capital letters to denote references into the dataflow store.) A mutable pointer consists of two parts: its *name*, which is a value, and its *content*, which is a reference into the dataflow store. The mutable store is *nonmonotonic* because pointer contents can be changed.

Threads contain statement sequences $S_i$ and communicate through shared references into the dataflow store. We say a thread is *ready* if it can execute its first statement. Threads are *dataflow* threads, i.e., a thread becomes ready when the first statement's needed arguments are bound. If a needed argument is unbound then the thread automatically blocks until the argument is bound. Since the dataflow store is monotonic, a thread that is ready will stay ready at least until it executes its first statement. The system guarantees weak fairness, which implies that a ready thread will eventually execute its first statement.

In a distributed setting, each mutable pointer exists on exactly one site at any instant. Both mutable pointers and dataflow variables may be referenced from more than one site. Each site has part of the dataflow store and part of the mutable store, corresponding to the variables and mutable pointers it references. The distributed protocol used to bind dataflow variables guarantees that all sites stay consistent with each other.

### *3.2 Reduction Rules*

Oz execution is given by an interleaving semantics defined in terms of reduction rules. The semantics of a statement is defined by the atomic reduction of rules written as follows:

$$\frac{S \;\|\; S'}{\sigma \;\|\; \sigma'} \quad C$$

where $C$ is a boolean condition, $S$ and $\sigma$ are the statement and the store *before* the reduction, and $S'$ and $\sigma'$ are the new statement and store *after* the reduction.

A rule becomes applicable for a given statement when the actual store matches the store given in the rule and the condition $C$ is true. Reduction of multiple threads is in general nondeterministic, since many threads can be ready. The effect of a reduction is to replace the current configuration $(S;\sigma)$ by a result configuration $(S';\sigma')$. Weak fairness between threads implies that a rule is guaranteed to reduce eventually if it is applicable and it refers to the first statement of a thread.

[b] Mutable pointers are called cells in [17].

### *3.3 Justification of the Model*

For the curious reader, we briefly justify the Oz execution model. This section is not needed to understand the article. See [4] for more information.

- **The store**. We observe that the monotonic and nonmonotonic stores are fundamentally different.
  - The monotonic store is easy to reason with [18], but it cannot easily express computations that interact with their environment. The monotonic store is given a dataflow behavior, which makes it more useful than a pure value store. In particular, dataflow variables are useful for concurrent programming.
  - The nonmonotonic store is needed for computations that interact with their environment. Mutable pointers make object references possible [19]. Without mutable pointers stream merging is necessary, which is cumbersome and inefficient [20]. Mutable pointers and dataflow variables together allow to express most concurrent programming idioms in a simple way [16].
- **Procedures**. Higher-order procedures with lexical scoping are the basic mechanism for building abstractions. This mechanism is not limited to pure functional languages; it is also useful in a concurrent language with a nonmonotonic store. In particular, higher-order procedures and mutable pointers are the foundation of the object system [21].
- **Threads**. Explicit threads facilitate debugging, reasoning about liveness properties (such as termination), and exception handling. Originally, the Oz language had implicit concurrency, where execution order was constrained by data dependencies between single instructions [15]. We found this to be inefficient, hard to program, and hard to reason about. Since Oz 2, all concurrency is introduced by explicit thread creation [22].

## 4 Semantics for Fault-Free Systems

We first give the centralized and distributed semantics of mutable pointers in a language whose implementation never has faults. Then we outline the distributed protocol that implements the distributed semantics.

For both mutable pointers and dataflow variables, we have given proofs that the distributed protocol implements the centralized semantics. The proofs are outside the scope of this article; for mutable pointers see [17] and for dataflow variables see [16]. The proofs use the fact that there exists a mapping

from any distributed execution to a centralized execution. This mapping can be used to prove properties about the distributed execution in terms of the centralized execution. In particular, one can prove that any distributed execution corresponds to a correct centralized execution. This means that the system is *network transparent*: the centralized language semantics remains valid in the distributed system.

### *4.1 Fault-Free Centralized Semantics*

A mutable pointer is a pair `n:W` of a unique name `n`, which is a constant, together with `W`, which is a reference into the dataflow store. The mutable pointer is referred to by its name `n`; the pair `n:W` is referred to indirectly through the exchange operation. Exchange does an atomic read-and-write. Exchange is written as the statement `{Exchange X Y Z}`; its semantics is defined by the following reduction rule:

$$\frac{\texttt{\{Exchange X Y Z\}} \;\|\; \texttt{Y=W}}{\texttt{X=n} \wedge \texttt{n:W} \wedge \sigma \;\|\; \texttt{X=n} \wedge \texttt{n:Z} \wedge \sigma}$$

This rule is reducible when its first argument `X` refers to the name of a mutable pointer. It reduces to the new statement `Y=W`, which is a binding request that will give access to the old content `W` by means of `Y`. The mutable pointer's new content is `Z`.

### *4.2 Fault-Free Distributed Semantics*

The distributed semantics is a refinement of the language semantics. The refinement specifies how the reduction is partitioned among multiple sites. We annotate each statement, store binding, and mutable pointer with the site it is on. For example, to denote that statement $S$ exists in a thread on site $i$, we write $S_i$. A binding `X=t` on site $i$ is denoted $(\texttt{X=t})_i$.

We assume that the mutable pointer's name (`n`) can be known at many sites, but that the mutable pointer itself (`n:W`) exists on exactly one site at any instant. We write $(\texttt{n:W})_j$ to denote that the pointer exists on site $j$. If another site than $j$ wants to update the pointer, then network operations are needed. In terms of distributed semantics, these operations are not mentioned explicitly; the reduction of exchange does an atomic transition of the pointer from one site to another.

The distributed semantics of exchange is therefore given by the following reduction rule:

$$\frac{\texttt{\{Exchange X Y Z\}}_i \;\|\; (\texttt{Y=W})_i}{(\texttt{X=n})_i \wedge (\texttt{n:W})_j \wedge \sigma \;\|\; (\texttt{X=n})_i \wedge (\texttt{n:Z})_i \wedge \sigma}$$

This rule states that, assuming the mutable pointer is on site $j$ and the exchange is on site $i$, then after the reduction the mutable pointer will be on site $i$, the old content of the pointer will be accessible through Y, and the new content of the pointer will be Z.

If $i \neq j$ then this rule implies that the system must send messages between sites to atomically bring the mutable pointer from site $j$ to site $i$. Any additional exchanges on site $i$ will be purely local, i.e., require no access to any site other than $i$.

### *4.3 Fault-Free Distributed Protocol*

The protocol that controls the mobility of the mutable pointer is very simple. Here we summarize the protocol briefly; see [17] for a formal definition and correctness proof. A mutable pointer has a fixed *home site* and a variable *current site*. The home site is the site at which the pointer was originally created. The home site is used to serialize requests for the pointer. The current site is the site that is hosting the pointer at a particular moment in time. A site requests the pointer by sending a message to the home site. The latter then sends a forwarding message to the current site.[c] When the pointer arrives there, the current site forwards it to the requesting site, which becomes the new current site.

We summarize the network operations needed to implement the distributed exchange rule according to this protocol:

- If the exchange is done on the current site, then no network operations are needed.

- If the exchange is done on the home site or the current site is the home site, then two messages are needed: one from the exchange to the current site, and one in the opposite direction.

- In any other case, moving the pointer requires three messages to be passed between sites. One from the exchange site to the home site, one from the home site to the current site, and finally one from the current site to the exchange site.

This gives the programmer a simple and predictable mental model of what happens on the network. We say that the system is *network aware* [6].

[c] More precisely, to the last site that will eventually become the current site.

## 5 Semantics for Fault-Prone Systems

This section gives the centralized and distributed semantics of mutable pointers when the failure model is included. The failure model is designed to be simple and yet cover the vast majority of failures that occur in practice for Internet applications. There are two complementary views of the failure model. The first view explains the network-layer faults, i.e., the system faults that are recognized by the Mozart implementation (Section 5.1). The second view explains how these faults show up at the Oz language level, i.e., in terms of mutable pointers (Section 5.2).

### *5.1 Network-Layer Failure Model*

The network layer recognizes two kinds of system faults:

- Permanent site failures that are detectable by other sites. I.e., a site from some point onwards reduces no more rules, and this fact is eventually known by any other site that attempts to contact the failed site. In the dependability community, this fault state is known as "fail-stop" or "fail-silent with failure detection" [23].

- Temporary network inactivity between a pair of sites. This fault state is characterized by a lack of precise knowledge of what the fault is and whether it will go away. Network inactivity may be caused by a failure of part of the network or of other sites. Network problems are always temporary, since it is always possible at least in principle to repair the network. The Mozart system is careful not to store any state in the network itself. This means that messages are never lost due to temporary inactivity. Furthermore, if the TCP protocol closes the connection, then the system actively tries to reestablish a connection.

In general, the network-layer faults that are detectable are limited by which network protocols the system uses. In the first Mozart release, sites are operating system processes and the network is the Internet with the TCP protocol family [24]. This means that there are cases in which we cannot say anything for certain; these cases are covered by a temporary inactivity state.

### *5.2 Language-Level Failure Model*

At the language level, fault states are defined with respect to both a language entity and a particular site. In general, fault states are possible for any

language entity [25]. In this article, we only consider the mutable pointer. A language entity is always in one of three fault states on a given site:

- The entity works normally (local fault state `none`).
- The entity is currently not working (local fault state `tempFail`). This is because a remote site crucial to the entity is currently unreachable due to a network or site problem. This fault state may go away, but this is not guaranteed.
- The entity is permanently not working (local fault state `permFail`). This is because a site crucial to the entity has crashed and this is detected. This fault state is permanent.

It is the responsibility of the distributed protocol implementing the mutable pointer to do the translation from network-layer faults to distribution-layer (i.e., language-level) faults.

### *5.3 More on Temporary Faults*

The fault state `tempFail` exists to allow the application to react quickly to temporary communication problems between sites. It is raised by the system as soon as a communication problem is recognized. It is therefore fundamentally different from a time out. For example, TCP gives a time out after some minutes. The TCP time out is very long, approximating infinity from the viewpoint of the network connection. After the time out, one can be relatively sure that the connection is no longer working.

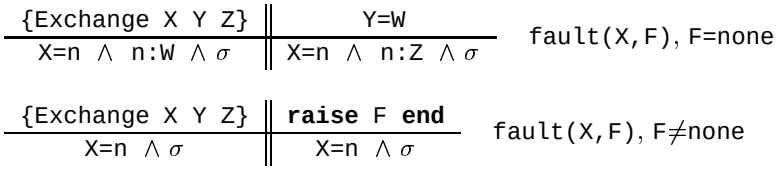
The purpose of `tempFail` is to inform the application of network problems, not to mark the end of a connection. For example, an application might be connected to a server. If there are problems with this server, the application would like to be informed quickly so that it can try connecting to another server. A `tempFail` fault state will therefore be relatively frequent, much more frequent than a time out. In most cases, a `tempFail` fault state will eventually go away.

It is possible for a `tempFail` state to last forever. For example, if a user disconnects the network connection of a laptop machine, then only he or she knows whether the problem is permanent. The application cannot in general know this. The decision whether to continue waiting or to stop the wait can cut through all layers of the system to appear at the top-most layer (i.e., the user). The application might then pop up a window to ask the user whether to continue waiting or not. The important thing is that the network layer

does not make this decision; the application is completely free to decide or to let the user decide.

### *5.4 Fault-Prone Centralized Semantics*

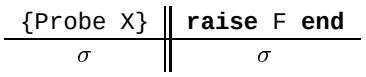
The exchange operation extended with the failure model has the following centralized semantics:




In these rules, `fault(X,F)` is a relation that gives a current fault state of `X`. Possible values of `F` are `none`, `tempFail`, and `permFail`. The operation **`raise F end`** raises an exception `F` in the current thread. It is part of the kernel language and has the standard dynamic scope semantics of exceptions [26].

What these rules say is that an exchange operation has two possible results: either do the exchange correctly, or do nothing and raise an exception describing an existing fault state. If an exception is raised then the application is free to retry the operation. If the fault is `tempFail`, then if the fault goes away, the retry will succeed.

The exchange operation lets us do synchronous failure detection, i.e., to detect a failure when we attempt an operation. In many cases it is useful to have asynchronous failure detection as well, i.e., a site may want to be notified of a fault as soon as possible even though it is not currently using the mutable pointer. To do this, we introduce the operation `Probe`, with the property that it never completes during normal operation. The probe operation exists for all language entities, not just mutable pointers. It is defined by the following rule:




To do asynchronous failure detection, it suffices to invoke a `Probe` in its own thread. This is practical in Mozart because of its lightweight threads.

### *5.5 Fault-Prone Distributed Semantics*

We saw in the previous section how the exchange operation behaves at the language level if the underlying system is faulty. At the language level, the

behavior is not very useful, since we do not know at which site the fault was detected. However, at the distribution level, the behavior of exchange is definitely useful, since it tells us on which site the fault occurs. The distributed semantics of exchange and probe are:

$$\frac{\{\texttt{Exchange X Y Z}\}_i \;\|\; (\texttt{Y=W})_i}{(\texttt{X=n})_i \wedge (\texttt{n:W})_j \wedge \sigma \;\|\; (\texttt{X=n})_i \wedge (\texttt{n:Z})_i \wedge \sigma} \quad \texttt{fault(X,F)}_i, \texttt{F=none}$$

$$\frac{\{\texttt{Exchange X Y Z}\}_i \;\|\; (\textbf{raise F end})_i}{(\texttt{X=n})_i \wedge \sigma \;\|\; (\texttt{X=n})_i \wedge \sigma} \quad \texttt{fault(X,F)}_i, \texttt{F}\neq\texttt{none}$$

$$\frac{\{\texttt{Probe X}\}_i \;\|\; (\textbf{raise F end})_i}{\sigma \;\|\; \sigma} \quad \texttt{fault(X,F)}_i, \texttt{F}\neq\texttt{none}$$

These three rules completely define the language interface for mutable pointers, taking into account the concerns of distribution structure and fault tolerance. Two questions remain to be answered:

1. Can these rules be implemented efficiently, i.e., with no overhead when there are no failures?
2. Do these rules suffice to build in Oz the fault tolerance abstractions we need?

The next two sections give answers to these questions.

## 6 Distributed Protocol with Failure Detection

The first question to answer is whether the three semantic rules of the preceding section are practical, i.e., can they be implemented efficiently, given our failure model? The only good way we know of answering this question is by actually implementing the protocol. This has been done in the Mozart system [27]. When there are no faults, the resulting protocol has essentially the same performance as the protocol of Section 4.3. The message latency of mobility is the same, but there is one extra message whose cost can be amortized [11]. The protocol also has the good property that no matter what permanent failures happen (either of the home site, the current site, or any other site), that each site making a request will eventually get correctly informed about the situation. This is expressed precisely by the following theorem (taken from [28], which also has a proof):

**Theorem 1 (Mutable pointer with failure detection)** *If the mutable pointer is requested at a site, then exactly one of the following three statements is eventually true:*

1. *The home site does not fail and the pointer is never lost (e.g., the pointer is never on a site when that site fails permanently). Then the requesting site will eventually receive the pointer exactly once. The exchange executes normally.*

2. *The home site does not fail and the pointer is lost before it reaches the requesting site. Then the requesting site will never receive the pointer, but it will eventually receive notification from the home site that the pointer is lost. The exchange or probe operation raises an exception.*

3. *The home site fails. Then the requesting site is eventually notified of this. If it does not have the pointer, then it knows that it will never receive it. If it has the pointer, then it knows it will keep it forever. The exchange or probe operation raises an exception.*

## 7 Building Abstractions for Fault Tolerance

The second question is whether the three rules of Section 5.5 suffice to build the fault tolerance abstractions we need. So far, we only have a partial answer to this question. We have built the following three abstractions:

1. The handler model, which is part of Mozart [25]. When an operation is attempted on a mutable pointer, then if the operation cannot be completed, it is replaced by a call to a user-defined procedure, called *handler*. The handler can implement arbitrary retries and time outs. The handler model gives a slightly higher level of abstraction than Section 5.5.

2. A replicated object abstraction. An object's state is replicated on several sites, and a program implements a coherence protocol between the replicas. We have built and tested a simple prototype of this abstraction. This abstraction does not appear to be immediately useful to distributed applications, so we are not continuing its development.

3. Transactional fault tolerance of an object store [12]. The store is similar to the store built in the PerDiS project [29]. Our store provides a more flexible interface that allows various degrees of synchronization among sites, from fully asynchronous (complete decoupling of local operations from the network; some speculative computation possible) to fully synchronous (no speculative computation; greater dependence on the network). Fault tolerance is realized by implementing a single object as $n$ mutable pointers, each on a different site. Any site can submit a transaction, which is a sequence of state updates to a set of store objects. As long as one

site survives, then the object store survives. We are currently testing a prototype implementation of the object store.

We find that the transactional object store is a good abstraction for collaborative applications. We are integrating it into a collaborative graphic editor [30] and we plan to build a reliable dictionary service with it.

## 8 Conclusions

We have presented and justified the semantics for mutable pointers that is used in the Mozart Programming System [27], which implements the Oz language. This semantics clearly separates the concerns of functionality, distribution structure, and fault tolerance. We give evidence that it is possible to provide a language interface to mutable pointers that lets one build nontrivial abstractions for fault tolerance within the language. Furthermore, the protocol that implements this interface does not lose performance compared to a protocol without failure detection.

However, these results do not solve the whole problem. We are continuing to build fault tolerance abstractions and applications that use them, to determine which primitive operations should go into the implementation. We are also working on a model that takes security into account.

## Acknowledgments

This article has profited greatly from discussions with Per Brand, Seif Haridi, and Raphaël Collet. Some of the basic ideas on language-based fault tolerance are originally due to Per Brand and Seif Haridi. The Mozart system's distribution and network layers were implemented by Per Brand and Erik Klintskog. The transactional object store is being implemented by Iliès Alouini. Thanks to Phil Trinder and Kevin Hammond for fruitful discussions at the PDSIA '99 workshop. Thanks to Luc Onana for his insightful comments. This research is partially financed by the Walloon Region of Belgium.

## References

1. Alan Pope. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison-Wesley, 1997. See also `http://www.omg.org`.
2. Sun Microsystems. *The Java Series*. Sun Microsystems, Mountain View, Calif., 1996. See also `http://www.javasoft.com`.

3. Gerard Tel. *An Introduction to Distributed Algorithms.* Cambridge University Press, Cambridge, United Kingdom, 1994.
4. Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3), May 1998.
5. Jean-Claude Laprie. Dependability: A unifying concept for reliable computing and fault tolerance. In *7th International Conference on Distributed Computing Systems*, pages 129–146, September 1987.
6. Luca Cardelli. A language with distributed scope. In *Principles of Programming Languages (POPL)*, pages 286–297, January 1995.
7. Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.
8. Richard D. Schlichting and Vicraj T. Thomas. Programming language support for writing fault-tolerant distributed software. *IEEE Transactions on Computers*, 44(2):203–212, February 1995.
9. Jean-Charles Fabre and Tanguy Pérennou. A metaobject architecture for fault tolerant distributed systems: The FRIENDS approach. Technical report, Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS), Toulouse, January 1997.
10. John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.
11. Peter Van Roy, Per Brand, Seif Haridi, and Raphaël Collet. A lightweight reliable object migration protocol. Lecture Notes in Computer Science, vol. 1686. Springer Verlag, October 1999.
12. Iliès Alouini and Peter Van Roy. An open distributed fault-tolerant transactional store in Mozart. In preparation, 2000.
13. Gert Smolka. Problem solving with constraints and programming. *ACM Computing Surveys*, 28(4es), December 1996. Electronic Section.
14. Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 519–533, Schloß Hagenberg, Austria, October 1997. Springer-Verlag.
15. Gert Smolka. The Oz programming model. In *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.
16. Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3), May 1999.

17. Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.
18. Vijay A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
19. Sverker Janson, Johan Montelius, and Seif Haridi. Ports for objects in concurrent logic programs. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.
20. Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.
21. Martin Henz. *Objects for Concurrent Constraint Programming*, volume 426 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, November 1997.
22. DFKI Oz version 2.0, February 1998. Available at `http://www.ps.uni-sb.de`.
23. Shivakant Mishra and Richard D. Schlichting. Abstractions for constructing dependable distributed systems. Technical report, Dept. of Computer Science, University of Arizona, August 1992. TR 92-19.
24. Douglas E. Comer. *Internetworking with TCP/IP. Vol. 1: Principles, Protocols, and Architecture*. Prentice-Hall, Englewood Cliffs, N.J., 1995.
25. Peter Van Roy, Seif Haridi, and Per Brand. Distributed programming in Mozart – A tutorial introduction. Technical report, 1999. In Mozart documentation, available at `http://www.mozart-oz.org`.
26. Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass, 1985.
27. Mozart Consortium (DFKI, SICS, UCL, UdS). The Mozart programming system (Oz 3), January 1999. Available at `http://www.mozart-oz.org`.
28. Per Brand, Peter Van Roy, Raphaël Collet, and Erik Klintskog. A reliable mobile state protocol. In preparation, 1999.
29. Paulo Ferreira, Marc Shapiro, Xavier Blondel, Olivier Fambon, João Garcia, Sytse Kloosterman, Nicolas Richer, Marcus Roberts, Fadi Sandakly, George Coulouris, Jean Dollimore, Paulo Guedes, Daniel Hagimont, and Sacha Krakowiak. PerDiS: design, implementation, and use of a PERsistent DIstributed Store. Technical Report 3525, INRIA, October 1998.
30. Donatien Grolaux. Editeur graphique réparti basé sur un modèle transactionnel (A distributed graphic editor based on a transactional model) (in French). Technical report, Université catholique de Louvain, June 1998. Mémoire de fin d'études (Master's project).