PERDIO

Persistent and Distributed Programming in Oz

Project Proposal

March 2, 1995

Gert Smolka
Christian Schulte
Peter Van Roy

German Research Center for
Artificial Intelligence (DFKI GmbH)

Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
Email: {smolka,schulte,vanroy}@dfki.uni-sb.de

Contents

1 Outline

Computer systems are undergoing a revolution. Twenty years ago, they were centralized, isolated, and expensive. Today, they are distributed, networked, and inexpensive. However, advances in software construction have failed to keep pace with advances in hardware. To a large extent, this is a consequence of the fact that current programming languages were conceived for sequential and centralized programming. Distributed systems, however, need radically different software than centralized systems.

The basic problem with current programming languages is that they delegate two basic functionalities to the underlying operating system, where they cannot be accommodated at a sufficiently high level. These functionalities concern the persistent storage of data (i.e., files) and the creation and coordination of possibly distributed computational agents (i.e., processes). Delegating these functionalities to the operating system has the severe disadvantage that the data abstractions (e.g., objects) of the programming language cannot be used for persistent storage or for communication between agents. Due to these discontinuities, the construction of software involving distribution and persistence cannot fully exploit the benefits offered by current programming languages.

The PERDIO project will develop a high-level programming system for networked computers, called Perdio in the following. Perdio will assist the cost-efficient construction of open, distributed, fault-tolerant, reactive, and knowledge-intensive software systems. A single Perdio program will be able to create multiple computations that spread over the network, and computations created by separate Perdio programs will be able to connect transparently. Perdio computations will be able to make use of persistent stores that can hold all data abstractions. It will be possible to access data structures without being aware of whether they are active in primary memory or passive in secondary storage.

Having both distribution and persistence in a single system gives synergy. The multitude of separate models (programming-language model, long-term storage model, distribution model) is replaced by a single model. Where previously explicit code had to be written to move data between models, in the single model none is needed. Where previously augmenting a program with the abilities of another model required major effort (e.g., making a program distributed), in the single model, a program can be written with only the single model in mind, and with minor changes it will have the abilities of the other models. The combination of persistence and distribution is of particular importance for the construction of fault-tolerant software systems.

Perdio will combine network-transparency with network-awareness. Network-transparency means that the semantics of computations does not depend on the site they execute on, and that the possible interconnections between two computations do not depend on whether they execute on the same or on different sites. Network-awareness means that distribution is achieved by explicit acts that give full control over communication patterns. While network-transparency makes it easy to write distributed software, network-awareness makes it possible to avoid undesirable network traffic and to fully plan the exploitation of the resources of a distributed computing system.

Perdio will be implemented on Unix-based workstations connected through standard Internet protocols. Perdio will spawn a world-wide computational universe that is organized at a much higher level of abstraction than the current Unix/Internet technology. Users without special system privileges will be able to install Perdio on their workstations and to transparently enter the Perdio universe through the Internet. We will experiment with scenarios from mobile computing and start work on an environment for collaborative and distributed program development.

Perdio will be based on Oz, a higher-order concurrent constraint language developed in a current project at the DFKI. Oz is already used inside and outside of the DFKI for applications that require processing of complex symbolic representations, organization into multiple agents, and soft real-time control. Oz's primary structuring concept for concurrent programming is the object. Objects combine stateful data abstraction with mutual exclusion and synchronization. Oz is based on logic variables and constraints, thus realizing an expressive and straightforward form of concurrent control that originated with concurrent logic programming. Oz offers the problem-solving capabilities of constraint logic programming by means of a higher-order search combinator spawning a subordinated computation space.

A key advantage of Oz is its simple yet expressive formal computation model. Objects and other programming abstractions are obtained from a small set of orthogonal primitives. Oz is in fact the first language whose concurrent object system is constructed in a principled and formal manner. Once distribution and persistence issues are understood for the basic computation model underlying Oz, they will extend readily to the rest of the language. This holds true both for conceptual and for implementational concerns. Oz's problem-solving capabilities come as a conservative extension to the rest of the language and are not expected to interfere with the extension to distribution and persistence. The main reason we were able to construct an efficient and stable implementation of Oz with so few resources is in fact the simplicity gained from Oz's computation model and the concomitant reduction to a lean kernel language.

First applications of Oz include multi-agent scenarios, simulations, scheduling, placement, configuration, graphical user interfaces, natural language processing, and virtual reality. There is no other concurrent language combining a rich object system with advanced features for symbolic processing and problem solving. The extension of Oz to distributed and persistent programming will yield a powerful and unparalleled programming tool showing the full potential of the concepts underlying Oz. Given the widely recognized fact that the current tools for distributed programming are primitive and unnecessarily complex, there is the chance that Perdio will have an impact at the very center of the future of computing.

2 Approach and Objectives

The main goal of PERDIO is to design, implement and explore a high-level programming language for networked workstations, called Perdio in the following. The workstations are assumed to run Unix-like operating systems, and to communicate through the standard Internet protocols (e.g., TCP/IP). The network may consist of high-speed local area networks and slower wide area networks (see Figure 1).

Perdio will support the cost-effective construction of open, distributed, fault-tolerant, reactive, and knowledge-intensive software systems. A single Perdio program will be able to create multiple computations that spread over the network, and computations created by separate Perdio programs will be able to connect transparently. Perdio computations will be able to make use of persistent stores that can hold all data abstractions. It will be possible to access data structures without being aware of whether they are active in primary memory or passive in secondary storage.

Perdio will combine network-transparency with network-awareness. Network-transparency means that the semantics of computations does not depend on the site they execute on, and that the possible interconnections between two computations do not depend on whether they execute on the same or on different sites. Network-awareness means that distribution is achieved by explicit acts that give full control over communication patterns. While network-transparency makes it easy to write distributed software, network-awareness makes it possible to avoid undesirable network traffic and to fully

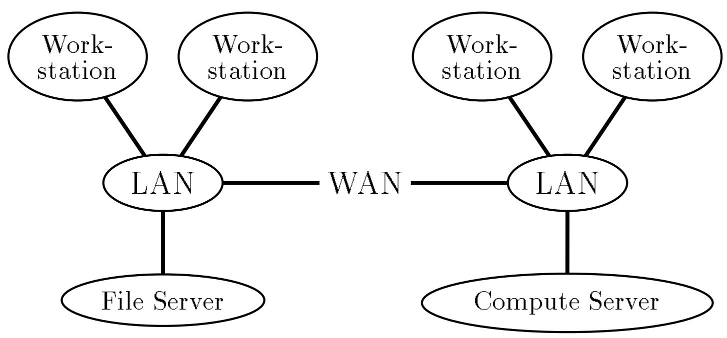

Figure 1: Networked workstations.

Applications

—

Perdio

—

C/Unix/Internet

—

Hardware

Figure 2: Perdio is a high-level platform for realizing distributed applications.

plan the exploitation of the resources of a distributed computing system.

Perdio will be based on Oz, a higher-order concurrent constraint language developed in a current project at the DFKI.

Perdio will be designed as an intermediate layer bridging the wide gap between the structures realized by the established C/Unix technology and the structures needed for applications (see Figure 2). With Perdio it will be possible to develop and maintain distributed software with drastically less programming effort than would be required if C/Unix were used. On the other hand, the C/Unix technology provides a portable and powerful base for building Perdio. Building Perdio from scratch would be unrealistic and wasteful because of the resulting complexity and nonportability. Perdio will be designed as an open system that easily connects with foreign software.

Perdio will spawn a world-wide computational universe that is organized at a much higher level of abstraction than the current Unix/Internet technology. Users without special system privileges will be able to install Perdio on their workstations and to transparently enter the Perdio universe through the Internet. We will experiment with scenarios from mobile computing and start work on an environment for collaborative and distributed program development.

2.1 Concurrency, Parallelism, and Distribution

It is important to clarify what we mean by concurrency, parallelism, and distribution.

Concurrency is a concept related to programming. A programming language is said to be concurrent if a program can create multiple computations that can communicate and synchronize with each other. Concurrency appeared with operating systems, which are much easier to understand as dynamic systems of communicating processes than as monolithic sequential programs. Note that this is the case although the majority of operating systems runs on a single processor.

Parallelism is a concept related to hardware. Hardware operations occur in parallel if they overlap in time. Operations in a program are concurrent if they could be, but need not be, executed in parallel. Operations that occur one after the other, ordered in time, are said to be sequential. We can have concurrency in a language without parallel hardware, and we can have parallel execution without concurrency in the language. Concurrency in the language increases the potential for parallelism, however.

Distribution is a concept relating to hardware and software. A distributed computing system is any networked collection of computers. A distributed program is a program that spawns communicating computations on more than one computer of a distributed system. Clearly, distribution presupposes both parallelism (at the level of hardware) and concurrency (at the level of software).

Parallel computing systems are interesting because they can offer more computing power than sequential computing systems. Distributed computing systems are interesting because they can offer more computing power, fault-tolerance, and support for geographically distributed applications. Fault-tolerance results from the fact that distributed software can be designed such that it continues to function if some of the computers of a distributed system have crashed or are cut-off by network failures. Examples of applications demanding distributed computing systems are the inventory system of a supermarket chain, computer-supported cooperative work, and the world-wide telephone system.

2.2 Concurrent Programming and Oz

Distributed programming is a special form of concurrent programming. To design a programming language for distributed programming, it is natural to start from a concurrent programming language. Given a sufficiently powerful concurrent language, the extension to distributed programming will hardly affect the language's appearance. If a distributed language is designed well, it should be possible to design systems without regard to the fact that they will run on networks of computers. Which parts will run on which computers can then be determined very late in the system's design, even if the final system runs on a network containing different computers and operating systems.

Most of today's programming languages are designed for sequential programming. These languages delegate the creation and coordination of multiple computational agents (i.e., processes) to the operating system, which has the severe disadvantage that the data abstractions (e.g., objects) of the programming language cannot be communicated between agents. Thus the benefits of these languages cannot be fully exploited for the construction of distributed software.

Lisp and Prolog, the two most popular languages used for programming in Artificial Intelligence, suffer from the same deficiency. Their support for concurrent and distributed applications is in fact inferior to what is offered in C and Unix (e.g., threads and signal handling).

The lack of powerful concurrent programming languages is the result of conceptual deficits. It was only in the early eighties that Hoare and Milner succeeded in constructing formal models for simplistic programming languages supporting communicating sequential processes. The first formal system supporting dynamic systems of concurrent agents (the Pi-calculus [Mil92]) appeared in the late eighties and is still considered avant-garde.

Perdio will be based on Oz, a high-level concurrent programming language suited for knowledge-intensive applications [Smo94b, HSW95, SSW94, SS94, Smo95, ST94]. Oz has been designed explicitly as a concurrent language. It is based on a new computation model combining concepts from functional programming, logic programming, and concurrency theory. The most important concepts of Oz relating to concurrent and distributed programming are the following:

1. **Concurrent Objects.** Objects are Oz's primary structuring concept for concurrent programming. Objects combine data abstraction and state with mutual exclusion and synchronization. Objects are created as instances of classes.

2. **Lexical Scoping.** Oz is based on lexical scoping. Lexical scoping is familiar from functional and Algol-like languages. Lexical scoping in Oz is general enough to express modules. Lexical scoping guarantees that

the semantics of procedures does not change when they are executed on different sites.

3. **Full Procedural Abstraction.** Procedures are first-class citizens and procedure definitions can contain further procedure definitions. Full procedural abstraction makes it easy to realize compute servers: a computational task can be packed as a procedure and then be shipped to another site for execution. Full procedural abstraction is found in functional languages and is lacking in Algol-like and logic languages.

4. **Full Compositionality.** Full compositionality generalizes the idea of full procedural abstraction to all language constructs. For instance, objects and classes are first-class citizens, and class definitions can contain procedure definitions and vice versa. Through full compositionality it is possible that a computational task shipped to another site can create objects of classes created elsewhere.

5. **Composite Values.** Oz provides records and tuples as stateless data structures called composite values. Stateful data structures can only be expressed through objects. The distinction between stateful and stateless data structures is important since a distributed implementation can share stateless data structures much more efficiently than stateful data structures.

6. **Abstract Store.** Oz computes over an abstract store containing abstract entities. This is standard in dynamically typed languages such as Scheme or Prolog. An abstract store is the prerequisite for important features such as network-transparency, persistent data structures, and automatic memory management including garbage collection. These features cannot be obtained for languages such as C or Modula-2, since they fail to fully hide the memory of the executing computer.[1]

[1]The lack of garbage collection in C++ means that the programmer is responsible for destroying objects. It has been observed that memory management issues often cut across natural abstraction boundaries. Thus it is not surprising that explicit memory management is a common source of many difficult to debug programming errors. If an object is freed while still being referenced, a later access may find the memory in an inconsistent state. The resulting dangling pointers are difficult to track down because they often cause errors in code which is far removed from the offending statement. Another major problem with explicit memory management is the memory leak. Memory leaks are caused when an object is not freed even though there are no references to it. Programs with this bug use up more and more memory until they crash.

7. **Nonblocking Threads.** Oz provides automatically synchronizing threads serving as virtual processors. If a thread encounters a task that cannot yet reduce, the task is automatically parallelized by moving it to a fresh thread.

8. **Logic Variables.** Logic variables are single assignment references that can be used before they are assigned a value. When being assigned a value, they become transparent. Tasks that require the value of a not-yet-assigned variable suspend and resume automatically when the value becomes available. Logic variables express dependencies between tasks without imposing an execution order. Logic variables are familiar from logic programming languages.

Most researchers would expect the above features in a high-level concurrent programming language, with the exception of logic variables. Surprisingly, Obliq (a recent language designed by Cardelli [Car94]) and Oz are the only languages with these features. Obliq does not have logic variables, however.

There is evidence that logic variables are the canonical form of reference for concurrent programming. For many synchronization and communication problems they are spectacularly expressive. To appreciate their expressive power, assume that an agent sends a message to another agent that will produce an answer to the message eventually, and that the sender cannot wait without further action until the answer arrives.[2] The interesting question raised by this scenario is how the sender synchronizes on the arrival of the answer. With logic variables, the solution is straightforward: before sending the message, the sender creates a fresh logic variable acting as placeholder for the answer and includes it in the message. The logic variable can now be used by the sender exactly as if it already were the answer. In case a future action of the sender needs to inspect the answer, the action will suspend and resume automatically once the answer becomes available.

2.3 The Oz Computation Model

The key advantage of using Oz is its simple yet expressive formal computation model [Smo94c, Smo94b, Smo95]. Objects and other programming abstractions are obtained from a small set of orthogonal primitives. Oz is

[2] Imagine you are ordering a pizza using a mobile phone while driving home in your car: You will have to drive home before you can receive the pizza.

in fact the first language whose concurrent object system is constructed in a principled and formal manner.

Oz is based on a kernel language called Oz-2 [Smo94c]. Oz-2 consists of a minimal set of orthogonal computational primitives, which can express all programming constructs of Oz, except the constructs providing for problem solving. Oz-2 can express higher-order functions, concurrent objects, and classes with multiple inheritance. Oz-2 can be extended with additional primitives so that it can express the problem-solving capabilities of Oz.

The decomposition of complexities realized by Oz's design together with the surprising simplicity of Oz-2 have been a breakthrough in the development of high-level concurrent programming languages. There are no other concurrent programming languages with comparable functionality and simplicity of design. Oz-2 is in fact a unified programming model combining essential concepts from functional programming, logic programming, and concurrency theory.

The simplicity of Oz-2 is the main reason we believe that we can make a breakthrough in the development of high-level concurrent programming languages providing for distribution and persistence. If distribution and persistence are understood for Oz-2, they will be understood for all of Oz. This holds true both for conceptual and implementational issues.

We will outline the Oz-2 computation model to give an idea of how we want to achieve distribution and persistence. We will make no effort to illustrate programming in Oz.[3] Rather, we will concentrate on Oz-2's abstract store and outline how it can be adapted to distribution and persistence.

Computation in Oz-2 takes place in a computation space hosting a store and a finite number of expressions $T_1, \ldots, T_n$ called tasks.[4]

$$\begin{array}{ccccc} & & \text{Store} & & \\ & / & & \backslash & \\ \mathrm{T}_1 & & \cdots & & \mathrm{T}_n \end{array}$$

Tasks are partial programs obtained according to an abstract syntax. Computation proceeds by reduction of tasks. When a task is reduced, it either

[3]The document [Smo94c] is an introduction to programming in Oz.

[4]In other papers on Oz tasks are called actors. Since tasks are reduced by threads and are not active on there own, calling them actors might be misleading. Although Oz does not do it, there is the possibility to model tasks as active entities.

creates new tasks or writes information into the store. When a task is reduced, it disappears. A task can only be reduced once the store contains sufficient information. Reducibility of tasks is a monotonic property: once a task becomes reducible, it remains reducible if other tasks are reduced before it. Reduction of tasks is the smallest unit of work.

The store consists of three components: The constraint store, which is maintained according to a declarative semantics, the procedure store, which holds the procedures created during execution, and the cell store, which holds mutable references. The constraint store and the procedure store are monotonic (i.e., once information is entered, it cannot be retracted). We will not present these issues further here; they are discussed in detail in [Smo94c].

Tasks are organized into threads. A thread is a virtual sequential processor reducing a stack of tasks. A thread proceeds by reducing the topmost task of its stack, where newly created tasks are pushed on the stack in a fixed order. If a thread encounters a task that cannot yet reduce, the task is moved to a fresh thread so that the original thread can continue with the remaining tasks. This automatic parallelization (with respect to virtual processors) of not yet reducible tasks is a unique feature of Oz. A thread with two or more tasks can always advance by reducing or parallelizing its topmost task. It is guaranteed that all threads that can advance will advance (i.e., fairness).

2.4 The Graph Model of Oz

We will now present a model that captures the aspects of the Oz-2 computation model that are relevant for persistence and distribution. This so-called graph model is essentially an abstract storage model for Oz-2. As such, it says very little about actual programming in Oz (see [Smo94c] for this topic). It is also important to keep in mind that many of the programming abstractions of Oz (i.e., objects, classes) are not primitive in Oz-2, but are obtained from Oz-2's primitives.

An Oz-2 computation space can be seen as a directed graph featuring four types of nodes: tasks, determiners, variables, and cells. The subgraph given by the determiners, variables, and cells constitutes the store. Figure 3 shows a computation space with two tasks and a store consisting of five determiners, two cells, and two variables.

Determiners represent numbers, literals, tuples, records, and procedures (i.e., closures). Variables represent logic variables that have not yet been assigned

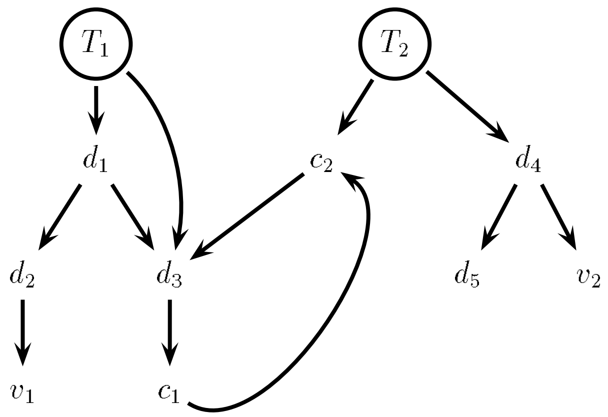

Figure 3: A computation space with two tasks T_1 and T_2.

a value. Cells represent mutable references. The links issuing from a determiner representing a tuple or record lead to its constituents, and the links issuing from a determiner representing a procedure lead to the global entities referred to by the procedure (i.e., lexical scoping). There are no links issuing from variables. There is always exactly one link issuing from a cell.

There are no links pointing to tasks. We say that a task depends on the nodes the links issuing from it are pointing to. In Figure 3, the task T_2 depends on the nodes c_2 (a cell) and d_4 (a determiner). Reduction of a task now amounts to graph rewriting, where every rewriting step is anchored at a task and only affects the nodes directly reachable from the task. Every task that does not depend directly on a variable is reducible. Tasks depending on a variable may have to wait until the variable is superimposed (see below).

The reduction of tasks may rewrite the store or create new tasks. In any case, the task being reduced disappears. The evolution of the store can be described by repeatedly applying the following rules:

1. A variable can be superimposed on another node taking with it all links leading to it. Superimposing a variable on another node makes the variable disappear. It is by superposition of variables that tasks depending on variables become reducible.

2. The store can acquire new subgraphs that are completely disconnected from the existing graph.

3. The link issuing from a cell can be redirected to another node.

The new subgraphs created by Rule (2) are connected with the rest of the graph by Rules (1) and (3). Determiners are static in that the links issuing from them cannot be redirected, and in that they cannot acquire new links issuing from them. Variables are monotonic in that they disappear once they are superimposed on another node. Cells are nonmonotonic in that their links can be redirected repeatedly.

The graph model assumes automatic garbage collection: Nodes that cannot be reached from tasks can be deleted. There is in fact a stronger garbage collection principle that provides for the deletion of certain irreducible tasks.

In the graph model, concurrency amounts to the presence of more than one task. All synchronization between tasks is accomplished through variables. The presence of logic variables is one of the most essential features of the Oz-2 computation.

In the following sections, we outline our basic ideas for extending Oz to persistence and distribution. It should be understood that we are presenting preliminary ideas illustrating the potential of the proposed research. The actual research developing these ideas into something complete and useful still needs to be done, of course.

2.5 Distribution

We have modelled computation spaces as directed graphs. Distribution means that the nodes of the graph may reside on different sites. We assume that every node resides on exactly one site. A nondistributed computation space is a special case of a distributed computation space, where all nodes reside at the same site. Links that connect nodes at different sites are called remote links, and links connecting nodes at the same site are called local links. Tasks may depend on nodes at other sites. Threads are not distributed, that is, all tasks of a thread must reside at the same site. Except for speed and assignment to sites, a distributed computation space evolves like a nondistributed computation space. This fact is referred to as network-transparency.

We distinguish between mobile and immobile nodes. Mobile nodes can move between sites, and immobile nodes remain fixed to one site. In our model, we assume that tasks and determiners are immobile, while variables and cells are mobile.

When a task reduces, it may create new nodes. We assume that all nodes created by the reduction of a task are allocated at the site of the creating task. This means that a computation has the tendency to remain local.

The distributed computation model we have outlined so far can easily express communicating agents executing on different sites. Agents located at different sites can share data structures in the distributed store and can thus communicate through logic streams, which are incrementally constructed lists exploiting logic variables.[5] Since we also have cells, we can provide stable reference to remote agents.[6]

Since we have full procedural abstraction and lexical scoping, the outlined distributed computation model is very expressive. We can for instance easily express a compute server that runs at a given site and executes computations packaged as nullary procedures at other sites. Since we have lexical scoping and network transparency, such packaged computations can take all their "live" connections with them. It is possible to express stationary objects processing messages sent from elsewhere, as well as mobile objects travelling between the sites sending them messages. Given a compute server on a remote site, we can install any computation we would like to at this site.

So far we assumed that no node moves from one site to another. To reduce network traffic it seems however important to move nodes. Only cells and variables can be moved. While the movement of a cell must be explicitly requested (there will be a special fetch task to fetch a cell to the site of the task), variables are moved automatically by the storage management.

Cells are not used directly by the programmer. Rather, they serve to express high-level abstractions such as objects. Providing explicit migration control for cells makes it possible to experiment with different high-level abstractions and migration policies.

There is an additional automatic mechanism, called replication, that helps

[5] Bill Silverman has given the following analogy: a logic stream is like a Genie, from which you can ask one wish. What would you ask such a Genie? To have two more wishes! Then use one of them, and iterate.

[6] Cells make it possible to express ports [JMH93], which provide stable reference to logic streams. This means that we are relieved from explicitly maintaining logic streams, a complexity plaguing concurrent logic programming. These issues are discussed in detail in [Smo94c].

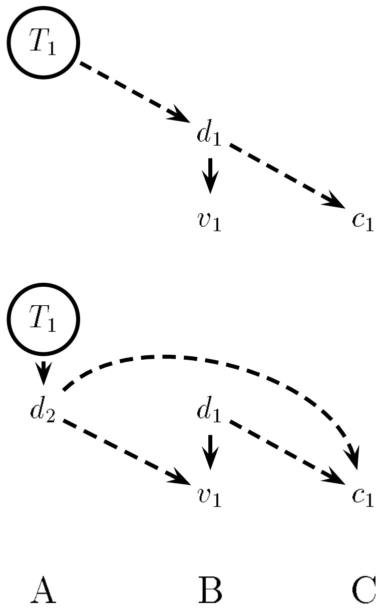

Figure 4: The determiner d_1 at site B is replicated as d_2 at site A.

to reduce network traffic. Replication copies determiners to other sites so that they can be pointed to locally.

If a determiner is pointed to by a remote link originating from a task, it will be replicated to the site of the task. Replication copies the determiner, where the links issuing from the determiner are also replicated. Replication is illustrated in Figure 4. Replication amounts to caching global determiners at a local site. Replication pays if a remote link is accessed more than once.

Replication is done automatically by the system if a task points to a remote determiner, or during garbage collection if a determiner is only pointed to by remote links.

It is up to the system to decide where it moves variables, subject to the restriction that variables can only be moved to sites from which they are pointed to. If a variable no longer pointed to from its current site, then garbage collection will eventually move it to a site from which it is pointed to.

2.6 Persistence

Computers have both volatile and persistent stores. While the content of volatile stores is lost when the computer is switched off or crashes, the content of persistent stores survives and can be used further once the computer is operational again. Operating systems provide persistent stores by means of file systems typically realized with disks. Almost all useful programs make use of persistent stores. The use of persistent stores is essential for fault-tolerant systems: If a computer crashes, it can reboot and continue with the data saved in its persistent stores.

Most of today's programming languages provide persistent stores only through an interface to the file system or to a database system. These forms of persistent store are very unsatisfactory since they don't match the data structures of the programming language. There is no way to store directly data structures containing cycles, procedures or objects. However, many applications, including CAD/CAM systems, office automation facilities, natural language processing systems, and software engineering environments, require persistent storage of such data structures. Atkinson [ABC+83] reports that a significant amount of programming effort is devoted to converting from file or data base formats into and out of program-internal formats.

There is a straightforward way to extend Oz-2 with a notion of persistence reminiscent of orthogonal persistence [ABC+83]. The store is extended with one distinguished node called the persistence root, which satisfies the invariant that no variables can be reached from it (so-called closedness). The substore consisting of all nodes reachable from the persistence root is called the persistent store, and its nodes are called persistent nodes. The persistent store survives when the computation space aborts or crashes.

If the persistent store contains cells, it is possible to update it by redirecting links issuing from persistent cells. Thus nodes that have not been persistent so far may become persistent, and nodes that have been persistent so far may lose this property. The operations updating persistent cells must be implemented as transactions, of course. This ensures that the persistent store is always in a consistent state, even after a system crash.

Persistent nodes are transparent in that they allow exactly the same operations as volatile nodes. The only visible difference is that updates of volatile cells can be done immediately, while updates of persistent cells must suspend until the subgraph they are requested to designate is closed (i.e., contains no variables).

Since objects and classes in Oz are defined by reduction to Oz-2, persistent objects and persistent classes come for free. The same holds for other programming abstractions defined by reduction to the kernel language. This issue illustrates the economy obtained by the reduction approach taken in Oz (i.e., defining a rich language by reduction to a lean kernel language comprising a carefully designed collection of orthogonal primitives).

From the primitive transactions obtained with persistent cells it is possible to program more structured persistent stores with more complex transactions. We expect that objects are the right abstraction for programming sophisticated persistent stores.

2.7 Persistence and Distribution

Next we outline how distribution affects persistence. We assume that every site comes with its own persistence root, and that every node reachable from the persistence root of a site must reside on this side. This means that persistent stores are local to sites, and that the persistent stores of different sites are disjoint. However, it is perfectly possible that a volatile node can point to persistent nodes at different sites. Distribution also affects the transaction behaviour of persistent cells in that their links can only be redirected to local and closed (i.e., variable-free) subgraphs of the store. While remote determiners can always be localized through automatic replication, remote cells cannot be localized automatically. In fact, persistent remote cells cannot be moved as long as they are persistent.

The outlined model seems to accommodate distributed persistence in a remarkably straightforward way. One reason for its simplicity is the fact that our model is inherently concurrent.[7] Research is however needed to find out how and whether it can be implemented efficiently. Moreover, it needs to be seen whether the proposed primitives can express the right high-level programming abstractions.

[7]The lack of inherent concurrency has been reported as the main reason for the failure of the Avalon/Common Lisp project at CMU, which aimed at a distributed and persistent programming language based on Common Lisp [Cla91].

2.8 Open Systems Architecture

The Internet is an open architecture. While it is operational, new sites and new networks can join, and connected sites and networks can become disconnected (e.g., since they have crashed or have been shut-off). The Internet is also a fault-tolerant architecture. If parts of it are disfunctional, the rest can still function.

We would like to have a similar open architecture for Perdio. Let us refer to the union of all simultaneously existing Perdio computation spaces as the Perdio universe. New Perdio sites should be able to connect transparently with other sites. If some Perdio site crashes or is shut-off, the remaining sites should be able to continue in a well-behaved manner. We will also consider boot servers, which restart Perdio sites from their persistent stores in case they have crashed. Scenarios we will experiment with include collaborative program development and mobile computing. It should be possible for a user to enter the Perdio universe from any Internet-connected computer and get transparent access to his environment (i.e., the environment is mobile rather than stationary at his home computer). Users will be able to send each other e-mails containing arbitrary Perdio data structures. For instance, a message may carry with it methods to print, view and listen to it. It will also be possible to communicate a complex lexical entry from a natural language processing application such that all its connections (e.g., to the underlying taxonomy) are preserved.

A Perdio site is a process executing on a computer. Several Perdio sites may execute simultaneously on a single computer. A Perdio file is a portable file containing a persistent Perdio store. Execution of a Perdio file creates a new Perdio site. The new site starts with a computation space consisting of the persistent store in the file and a single task applying a nullary procedure given by the persistence root. This can initiate arbitrary computations, which may modify the persistent store of the local site as well as of other sites. Modifications of a persistent store immediately result in modifications of the file associated with the site of the store. If a site exits, aborts or crashes, its persistent store as existing at this point in time survives in the file associated with the site. Thus Perdio sites can be restarted from the well-defined states given by their persistent stores.

Perdio applications may register with special boot servers to ensure that they are restarted after a shutdown or crash. A boot server periodically probes each of its registered applications to determine whether it is still running. This scenario shows that transparent persistence is a useful feature for constructing fault-tolerant software.

Due to the expressiveness of our model, an arbitrary connection pattern between two Perdio sites can always be initiated by a single remote link, provided the destination of the link is chosen well (e.g., a server object).

The basic mechanism for connecting two Perdio sites is as follows. One site creates a port (e.g., a Unix port) that can receive a link to a remote node, and that is identified in the Internet by a port number and the address of the executing computer. The port identification is then shipped to the other site (e.g., by phone, e-mail or some Internet channel), which then connects with the port and provides reference to one of its nodes.

From this primitive connection mechanism more elaborate connection disciplines can be bootstrapped. One possibility is to run special Perdio sites serving as operators. Operators register with the name servers of their host computers so that they can be contacted conveniently through the Internet (e.g., as `perdio-operator.dfki.uni-sb.de`). Perdio sites will know one or several Perdio operators and connect with them when they are started. Thus Perdio operators serve as the fixedpoints in the open Perdio universe.

As in operating systems and networks, security and protection are important issues in the Perdio universe. Here Perdio will offer completely new and exciting possibilities. Lexical scoping makes it possible to hide things completely by construction. Another relevant feature of Oz are names, which are dynamically created tickets that cannot be faked. Names are reminiscent of the capabilities used in Amoeba, a distributed operating system [TvRvS+90], but are more basic and universal in that they are not committed to a particular access policy. Based on names, lexical scoping, and procedural and object-based abstraction, it should be possible to express arbitrarily sophisticated access control schemes. We will design such schemes for several application scenarios, including mobile computing.

2.9 Agents and Groups

We envision an agent to be a group of computational activities that together provide a meaningful service. Agents should be first-class citizens presenting a well-defined interface to the outside. Given the right capabilities, it should be possible to suspend and resume the execution of an agent, to migrate an agent to other sites, to detect and handle termination and failure of agents, and to kill an agent. It should also be possible to control the computational resources (e.g., time and memory) available to an agent.

Currently, Oz cannot express agents. What is missing is a first-class grouping concept for tasks. Given such a grouping concept, different forms of agents could be expressed as high-level programming abstractions similar to the way objects are expressed in Oz.

Threads are groups of tasks, but they do not have first-class citizenship. Moreover, threads organize tasks with respect to control and not with respect to functional dependencies.

Perdio will extend Oz with a first-class grouping concept for tasks, called groups. Based on groups, Perdio will be able to express concepts such as agents, concurrent and distributed exception handling, resource control, and migration of tasks to other sites.

A group is a collection of threads. Groups are local to sites, that is, all tasks contained in the threads of a group must reside at the same site. The resulting hierarchy of grouping notions for tasks can be depicted as follows:

$$\text{universe} \longrightarrow \text{site} \longrightarrow \text{group} \longrightarrow \text{thread} \longrightarrow \text{task}$$

Given two groups, they are always disjoint. If a thread in a group creates a new group, the new group is moved outside. Groups are uniquely identified by names, which are first-class values.

The main operation on groups is reflection. Reflection turns a group (i.e., a multiset of tasks) into a procedure (i.e., a node in the store). Reflection of a group is done by removing all tasks of the group and creating a nullary procedure that, when applied, recreates the tasks and the subgroups of the group. Since the reflected group is a procedure (i.e., a determiner in the store), it can be applied at another site, which then means that the group has migrated from one site to another. To preserve important structural invariants, we will impose the restriction that a reflected group (i.e., a procedure) can be applied only once.

We have studied and implemented reflection of tasks before in the context of a higher-order search combinator [SS94, SSW94] realized in Oz. The results have been very encouraging, regarding practical, conceptual, as well as implementational issues. There is an elegant formal model of the higher-order search combinator.[8]

Groups with reflection are an expressive concept. We have already argued that groups can be migrated from one site to another. Migration of groups is useful if a computer in a networked system needs to be shut down for maintenance. Migration of groups may also be used to program load balancing strategies for workstation farms.

Concurrent exception handling is another important concept that can be expressed with groups and reflection. The basic idea is to have exception handling at the level of groups. To raise an exception for a group means to reflect this group and to communicate some node in the store to an exception handler installed for the group. Exception handlers can be expressed with the existing concepts. To provide for the construction of software that has some robustness against programming errors, exception handling should also allow to catch run-time errors.

Another issue that we want to address at the level of groups is explicit resource control. It should be possible to impose bounds on the computation time and the memory a group can use. Once a resource bound is exceeded, the group may be reflected and a supervisor may be informed. The supervisor may then decide whether to abort the group, to give it more resources, or to migrate it to a more resourceful computer. The issue of resource control definitely needs research. For instance, it is not clear how to measure the memory used by a group since the store is shared among groups.

Groups will also provide for explicit termination control. When a group terminates (i.e., its last task has been reduced without producing further tasks), it applies a nullary procedure given to it when it was created. This concept will make it straightforward to detect the termination of concurrent and possibly distributed computations.

As outlined, groups are not a hierarchical concept, that is, groups cannot have subgroups. We are confident that we nevertheless can express hierarchical dependencies at the level of agents.

[8]The search combinator is in fact one of those constructs of Oz that were conceived based on Oz's formal model.

2.10 Program Development

Developing a large software system is a complex collaborative task. Software evolves from a prototype to a production level system with frequent changes made by different programmers during implementation and testing. Once the software is released, changes are still needed when bugs are removed, the implementation of the software is improved, or new functionality is added as new requirements are perceived. It is crucial that programming systems support the software development process.

We would like to support the collaborative construction of Perdio programs through Perdio itself. Given the fact that Perdio supports orthogonal persistence and open distribution, it seems to be an excellent base for constructing a distributed environment for collaborative program construction.

There are a number of interesting research issues to be addressed here. For one thing, it is not clear how one can combine full compositionality and lexical scoping (e.g., static binding) with incremental program construction (e.g., dynamic binding). This issues are much easier to deal with in flat noncompositional languages such as Prolog or Smalltalk. Another issue is linguistic reflection [Cut92]: How can one model with Perdio a compiler that transforms a text with respect to a binding environment for its free identifiers into a computation installing new, possibly procedural nodes in the store?

3 State of the Art

In this section we review related systems with respect to the topics of the previous section. We first give a brief overview of work in distribution. We then give detailed case analyses of four languages that support distribution, namely Orca, Erlang, Facile, and Obliq, and we compare them to Perdio. Finally, we discuss persistence, groups, open systems architecture, and programming environments.

3.1 Distribution

Distributed systems are commonly defined as a collection of autonomous computers connected by some form of network [BST89]. A well-designed distributed system provides the illusion of a single virtual machine with greater resources, reliability, and speed than its components. There are two approaches to creating such an illusion: It can be *provided* by the operating system, or it can be *layered* on top of an operating system. The first approach results in a *distributed operating system*. Examples of such systems are Amoeba [TvRvS+90], Clouds [DLAR91], and Chorus [RAA+88]. Tanenbaum [Tan95] gives a good overview of distributed operating systems. Most of these systems are still in an experimental stage [CC91] and none has become popular except for Mach [BKLL93].

Layering distribution on top of an operating system has significant advantages. It does not require reinventing the world and it is much more portable. It can take advantage of standard operating systems such as Unix and standard network protocols such as TCP/IP. A layered system, when entered, can provide a uniform view of the world, just as a distributed operating system does.

A simple approach to layering is the *coordination model* [CG89, CG92]. The prime example is Linda, which provides a uniform global "tuple space" that can be accessed from any process's address space through a small set of basic operations. Linda layers have been added to various languages including Fortran and C. Only primitive data objects (numbers, strings) can be put in the tuple space.

Layered distributed systems can be built either by directly accessing network protocols to do message passing [Ste90b] or by using a remote procedure call (RPC) facility [BN84, Ste90b]. Libraries to support both techniques are available for many systems—Turcotte [Tur93] cites over seventy. Building a distributed system by directly accessing these libraries is possible but difficult and bug-prone. Recently approaches like PVM (Parallel Virtual Machine) [GBD+94] and MPI (Message-Passing Interface) [GLS94] have become popular. The OSF DCE (Open Software Foundation Distributed Computing Environment) standard is a comprehensive set of integrated components layered on top of C/Unix that supports RPC, security, a distributed file system, and various global services.

The above systems provide poor support for interchange of procedures and complex data between address spaces. When more elaborate data structures than numbers or strings need to be transferred, the marshalling and unmarshalling routines (for translating between data structures and sequences of bytes) have to be designed and coded separately. Furthermore, the passing of pointer-based structures is cumbersome and the passing of procedures (i.e., closures) is impossible. In our terminology, these systems are not lexically-scoped and do not provide full procedural abstraction. In most cases there is also no abstract store, i.e., no model of the contents of each address space and how data is transferred between address spaces.

Because distribution necessarily implies concurrency, a solution to distribution that solves these problems requires a language that is based on concurrency. Many such languages have been designed, including Emerald [JLHB88], Arjuna [SDP91, PSWL94], Concert/C [AGG+94], SR [AOC+88], and Argus [Lis88]. In the following we discuss four languages in detail, namely Orca, Erlang, Facile, and Obliq, that address some or all of the above issues, and we compare them to Perdio.

Each of these languages has defined its own terminology. In order to discuss them in a uniform way, we will use the following common terminology. A *thread* is a virtual sequential processor. A *process* is an address space containing one or more threads. A *site* is the unit of distribution in a distributed system. Usually, but not always, a single computer hosts one site and each site contains one process. A system is *network-transparent* if operations are performed in the same way regardless of where they execute or whether their data is on a local or remote site. A system is *network-aware* if it gives full control over communication patterns. A system has *open* distribution if it can dynamically merge with another running system of the same kind. A system has *closed* distribution if it can dynamically create remote sites, but cannot merge with another running system.

3.2 Orca

Orca [BST89, BKT92, TKB92, BK93], developed at the Vrije Universiteit of Amsterdam, is a language for implementing parallel applications on loosely-coupled distributed systems. Orca's designers recognize the disadvantages of extending an existing sequential model with message passing. The implementation is based on replication of shared objects and reliable broadcasting. Replication is used to decrease the cost to access shared data.

Orca is first-order and object-based. It does not have inheritance, but an object's state is encapsulated and can only be accessed by user-defined operations. Objects are passive and correspond roughly to monitors [Hoa74, BH75]. Orca is weakly lexically-scoped in that operations can refer to variables local to their object. Operations on objects are executed atomically, thus guaranteeing serialization. Operations are allowed to yield output. Additionally, execution of operations can be synchronized by giving explicit conditions.

To solve the data interchange problem, Orca modifies the set of standard imperative data types by introducing safe pointers (called "general graphs"). Thus Orca provides for an abstract store, which is however not automatically garbage collected. This implies that accessing an already deallocated graph node issues a runtime error and that it is possible to have memory leaks and dangling references.

Concurrency in Orca is provided by the possibility to explicitly create new threads. A thread may take any kind of data structure either as *input* parameter or as *shared* parameter. In the first case, the thread gets a copy of the actual parameter. Thus, Orca incorporates cloning of objects. In the latter case, the object will be shared between the parent thread and the child thread.

Closed distribution in Orca is fully transparent. A remote computation is started by creating a thread at a remote site. Access to an object shared between threads at different sites results in implicit network operations and hence violates network-awareness. Orca does not provide for open distribution. Sites are identified by integers that are statically mapped to network addresses. All sites must be homogeneous, i.e., must run the same platform. Orca does not provide support for fault-tolerant applications. However, in [KMBT92] an extension to the Orca system is given that achieves transparent fault-tolerance.

3.3 Erlang

Erlang [AVW93, Hau94] is a concurrent functional programming language developed by Ericsson, Sweden, and conservatively extended for distributed programming [Wik94]. It is being successfully used for large distributed telecommunications applications. It provides pragmatic solutions for distribution and continuous operation and has an object-based structure. It combines a first-order functional language with first-class communicating threads.

A thread is explicitly created and executes a user-supplied function upon creation. Each thread is identified by an unforgeable *Pid* (Process Identifier). Pid's are first-class and can be passed in messages and used in pattern matching. Messages are sent asynchronously to a Pid and are put in a queue called a *mailbox*. A thread with its associated mailbox forms an active object with its identity being the Pid of the thread. Messages are received by guarded commands and are filtered with pattern matching. Messages in the mailbox are examined oldest-first to determine whether they match a pattern in a waiting guard.

Erlang provides for open distribution. Erlang is a very loosely coupled system in which sites may come and go at any time. Closed distribution in Erlang is provided by spawning new sites where a user supplied thread is created. Thus, Erlang is fully network-aware. All data structures can be communicated to remote threads; with regard to this aspect distribution is transparent in Erlang. However, sending a message to a remote thread differs from sending a message to a local thread in that the site where the receiving thread is computing on has to be explicitly specified. Data sent to remote sites will be replicated.

There is extensive support for building fault-tolerant applications. Within a thread, the primitives *catch* and *throw* provide exception-handling ability. Threads can be explicitly *linked* to other threads and to remote sites. Links serve to coordinate termination and error-handling behaviour. If a thread or site terminates, normally or abnormally, then all threads directly linked to it are notified, either by termination (the default) or by a user-defined catch of the termination's cause.

Since Erlang is strictly first-order, it needs rather complicated code management in order to supply sites with the functions needed for thread creation. The code management used in Erlang is dynamic in nature, i.e., code for a module can be loaded and linked at run-time, replacing the code of a running module without stopping the system. This property is regarded as essential for long-lived applications such as telephone switching systems.

3.4 Facile

Facile [TLP+93, LT92] is a concurrent functional programming language developed at ECRC, Munich. It combines a strict functional language, Standard ML (SML) [MTH90], with a model of higher-order concurrent processes based on the Calculus of Communicating Systems (CCS) [Mil89]. Its implementation is based on the Standard ML of New Jersey implementation from AT&T Bell Laboratories.

Concurrency in Facile is accommodated by explicitly creating threads. A thread executes code, called *script*. In contrast to Erlang, a thread has no identity. A script contains expressions including all Standard ML expressions as well as expressions for communication, dynamic channel generation, and thread creation.

Threads communicate with each other over typed bidirectional *channels*. Primitives for communication over channels are fully synchronous, i.e., both sender and receiver are blocked until the communication is realized. Communication is one-way, i.e., no return value is passed. Facile can indeterministically wait for a message to arrive on one of multiple communication channels. All values, including channels and scripts, can be passed over channels.

The exception handling of SML is provided within a thread. Exceptions are local to a thread, and do not propagate across threads.

No additional primitives for handling references (i.e., SML pointers) are provided beyond those in Standard ML. In particular, there are no mutual exclusion primitives for references that are accessible from multiple threads. Mutual exclusion (e.g., semaphores) has to be programmed using channels and scripts. Thus, Facile offers synchronization mechanisms to access shared data structures at a low level of abstraction.

Closed distribution is achieved by spawning a remote site and creating a thread at the remote site. Threads running on different sites can communicate over channels. Facile has no inter-site exception handling but provides support for distributed communication with time-outs. Facile does not provide for open distribution.

Distribution in Facile is not fully transparent. All values sent over channels between two sites are automatically replicated. If an SML reference is sent, the current value of the reference will be sent to the receiver. Updates to references will be not made visible at other sites. Because Facile does replication only on sending, it is fully network-aware.

3.5 Obliq

Obliq [Car94] is a distributed object-oriented language with lexical scoping, developed at the Systems Research Center, Digital Equipment Corporation. The Obliq implementation is an interpreter based on the Modula-3 Network Objects package [CDG+92, BNOW94]. Obliq has been designed from the start as a distributed language.

Obliq is based on a sequential model that provides multiple threads at multiple sites. It solves the data interchange problem by means of lexical scoping. Objects have a fixed site, and are passive and stateful. Support is provided for serialized access to objects.

Obliq is network-transparent: computations are executed without knowledge of their site, and remotely referenced data is transferred if needed. Obliq is network-aware: objects and variables have a fixed site. Values (i.e., Perdio determiners) are transmitted to where they are needed. Object methods are executed by creating a new thread on the object's site. Procedures are executed locally.

Obliq provides explicit *aliases* to redirect references across the net. Aliases are useful to forward communications across the network. For example, the effect of object migration is obtained by cloning an object and redirecting references to the original object. Aliases provide some of the functionality of logic variables. The major disadvantage of aliases is that they must be programmed explicitly.

Obliq provides for open distribution. A computation can register a network reference to an object with a name server. The reference is then globally accessible. Another computation can obtain it and communicate directly to the first object.

Issue	Facile	Erlang	Orca	Obliq	Perdio
Network-transparency	yes/no	no	yes	yes	yes
Network-awareness	yes	yes	yes/no	yes	yes/no
Distribution	closed	open	closed	open	open
Exceptions	yes/no	yes	no	yes	yes
Higher-order	yes	no	no	yes	yes
Objects	active	active	passive	passive	active passive

Table 1: Comparison of four distributed languages with Perdio.

3.6 Comparison

This section gives a comparison of the languages presented above and Perdio. Table 1 gives the comparison with respect to key issues in concurrency and distribution. Fields holding "yes/no" are explained below.

Facile is network-transparent if references are not used. Sending messages in Erlang is not fully transparent, since it forces the programmer to state whether the receiving thread is local or remote. All other languages are fully transparent.

Each language is network-aware with respect to threads. With respect to data, Orca does not give full control over communication patterns due to the implicit handling of updates to shared objects. The same holds for Perdio, where access to determiners and variables might involve replication and migration on behalf of the system.

Orca does not support any form of exception handling. Exception handling in Facile is local to a single thread. The other languages propagate exceptions between threads, independent of the sites on which the threads reside.

Erlang and Orca are first-order languages. Therefore they require explicit code management on creation of remote threads. For Orca, this imposes the condition that all sites must be homogeneous.

The most fundamental difference between these languages is how they support concurrency. Distribution is a consequence of how concurrency is accommodated. Concurrency is based either on active objects or on passive objects. Active objects are based on the message-passing style as known from CSP [Hoa85] and CCS [Mil89]. In active objects, threads are bound directly to the data abstractions. Passive objects are based on shared data structures, as exemplified by monitors [Hoa74, BH75]. In passive objects, threads and data abstractions are orthogonal issues.

Communication with an active object can be either synchronous or asynchronous. Facile does synchronous communication, however it is one-way only. Erlang does asynchronous communication, i.e., messages are buffered. In order to supply output on asynchronous communication, logic variables are needed. Since Erlang does not offer logic variables, it does only one-way communication.

Access to passive objects is always synchronous. By having synchronous access it is easy to have both input and output. Obliq and Orca are both based on passive objects and provide for input and output.

Perdio has primitives to easily express all the different styles mentioned above. Thus, Perdio makes it possible to experiment with different high-level concurrent programming abstractions.

3.7 Persistence

Persistence is commonly defined as the ability of data to outlive an execution of a program, and possibly the lifetime of the program itself [Cla91]. Research in persistence has been approached from two different directions, namely *database languages* and *persistent programming languages* [Cla91]. The motivation for database languages is to make database functionality available within the programming language [AB87]. Database languages integrate ideas from the database model with traditional programming language features. In the sequel we restrict the discussion to persistent programming languages since they are directly relevant to Perdio.

Applications are the main motivation for the integration of persistence into programming languages. These applications include CAD/CAM systems, office automation facilities, and software engineering systems. Atkinson [ABC+83] claims that a significant amount of programming effort is devoted to converting data from database or file formats into and out of program internal formats. Early measurements show that as much as 30% of all code in data-intensive applications is devoted to this task [IBM78].

The most important characteristic of a persistent language is whether it supports *orthogonal* persistence, namely that data of *all* types can be made persistent. The first language providing orthogonal persistence was a derivative of S-Algol [CM82] called PS-Algol [ABC+83, AM85], together with its successor Napier88 [MBC+88]. Other languages providing orthogonal persistence are Galileo [ACO85], OPAL [BMO+89], and Tycoon [MMS94]. Non-orthogonal persistence views persistence as a property of particular data types, as exemplified by Amber [Car86], E [RC89], and Avalon/C++ [HW87].

A second important characteristic is how to declare a type or data structure as persistent. An early technique that is still popular in interactive programming systems is the saved-state approach: Common Lisp [Ste90a], Standard ML [MTH90], SICStus Prolog [CWA+93], and Smalltalk [GR83] allow to checkpoint the entire state of the system. In Galileo [ACO85], the evolution of the entire store is reflected in the persistent store, and all data becomes persistent that is accessible from the top level at program termination.

A third important issue is how to provide access to the persistent store. Either there is a single persistent store which is automatically visible at system start or multiple persistent stores exist which usually have to be opened by explicit acts. A single, transparent store is the simplest model. Multiple stores force the programmer to assume some of the responsibility of management.

Communication between the volatile and persistent store can be achieved in many ways. Explicit import and export of data from and to the persistent store is one possibility, as exemplified by the Amber language [Car86]. Fully transparent access requires the system to import and export data between volatile and persistent stores on its own behalf. Galileo [ACO85] and PS-Algol [ABC+83] follow this approach, as does Perdio.

Other issues involved are how to locate and manipulate persistent data. Different approaches are known, falling in the same classes as mentioned above: either transparency (e.g., S-Algol [ABC+83] and OPAL [BMO+89]), or a distinction is made between data in the persistent and volatile store (e.g., Amber [Car86]).

Concurrency in persistent programming languages has been considered mainly in the form of access control to the persistent store. Since the persistent store is usually quite large and several programs have the need to access it, some form of locking is required. Most persistent languages do not provide for concurrency. They simply restrict access to the persistent store so that no sharing arises. Examples of this approach are PS-Algol [ABC+83], Amber [Car86], and Galileo [ACO85].

It has been acknowledged that persistent languages accommodate easily for the construction of powerful programming environments [Rei86, HZ87, RWW89, HSS89, Cut92]. We will elaborate on this point further in Section 3.10.

3.8 Groups

In this section we review the issues which are applicable to groups in Perdio, namely resource control, termination, exception handling, and migration. In other languages there is no unifying concept to which these issues are applied.

Resource control is usually found at the level of the operating system, not at the programming language level. Unix-like operating systems provide for so-called "limits": the amount of primary memory, disk space, and runtime can be limited [Ste92]. Once a particular limit is exceeded by a running process, the course of action taken by the operating system is hard-wired and drastic: the process is terminated.

Termination of groups is normally called *distributed termination* and has been investigated in the context of languages that create large numbers of lightweight threads, such as dataflow languages and concurrent logic languages. In languages with logic variables, a popular algorithm for distributed termination is based on the short-circuit technique [Tic91, Sha89].

Exceptions in concurrent languages can be either caught only in the thread in which the exception has been raised (e.g., Facile [TLP+93]), or they can be propagated to other threads (e.g., Erlang [AVW93], Obliq [Car94]). It is important whether propagation of exceptions is done across sites, as done in Erlang and Obliq. For more information on exceptions, see also the previous sections on the respective programming languages.

In Erlang [AVW93] it is possible to build groups of threads incrementally by explicitly creating bidirectional links between pairs of threads. These groups are used both for exception handling and termination.

The entities to be migrated are usually objects. Emerald [JLHB88] is a distributed object-oriented language that provides both explicit and implicit object migration. Objects within other objects can be *attached* to the enclosing one. On migration of the enclosing object its attached objects are migrated as well. Implicit migration is generally regarded as too inefficient and difficult to implement [BST89]. In Obliq, objects are stationary. The effect of migration can be achieved by cloning them at another site and redirecting references to them. Cloning is provided only for single objects.

3.9 Open Systems Architecture

According to Tanenbaum [Tan95], an *open system* is one that is prepared to communicate with any other open system by using standard rules that govern the format, content, and meaning of the messages sent and received. These rules are formalized in what are called *protocols*. The distributed artificial intelligence community has long discussed open systems from the viewpoint of concurrent agents [Hew85, HI91].

The Internet defines the state of the art for widely-used open systems. Information services, e.g., gopher, WAIS (wide area information system), WWW (world-wide web) [LPJ+94], recognize fixed sets of formats. For example, the WWW recognizes hypertext formats, graphic formats, and the like. A WWW browser can enter the "WWW universe" through any Internet node by contacting a known operator. For instance, the widely used Mosaic browser [DKF94] will first contact the WWW operator at CERN (Geneva, Switzerland), if it is not otherwise customized.

Basic services built on top of the Internet include electronic mail [Pos82], the Usenet news system [KL86], telnet (connection to remote host) [PR83], and ftp (file transfer protocol) [PR85]. For a guide to the many other services see for instance [Ste90b, Der94, LPJ+94]. The basic services manipulate unstructured entities only, usually text (byte sequences).

Closely related to the Internet is the ISO OSI (Open Systems Interconnection) Reference Model [DZ83]. It defines seven levels of abstraction for communication between open systems.

Security in distributed open systems has attracted much research, for a fairly recent overview see [GS94]. Most distributed systems usually also include support for security, for instance the OSF DCE (Distributed Computing Environment) offers extensive support for security [Tan95]. Widely found in distributed systems are capabilities [DV66]. A capability is associated to some entity in a system, usually an object providing some service. It is a kind of ticket or key, which allows the holder of the capability to perform some (not necessarily all) operations on the particular entity. For instance, capabilities are used in the distributed operating systems Amoeba [MT86, TvRvS+90], Chorus [RAA+88], and Mach [BKLL93].

3.10 Program Development

This section surveys the work on program development environments from the viewpoint of persistence. We single out Napier88, Galileo, and Smalltalk as examples of languages that support a persistent programming environment.

Orthogonal persistence has major benefits for programming environments. First of all, the system is strongly simplified by the merger of the long-term storage model (database or file system) with the programming-language data model. This means for example that no code need be written to support mappings between these models, resulting in a substantial reduction in programming effort [ABC+83].

Secondly, if the language is higher-order and reflective as well as persistent, then programs can be manipulated as data, so the complete programming environment can be built without leaving the language. All tasks previously done with help from the file system (such as compiling, editing, and linking) are internalized. A major unresolved research problem in this area is how to combine full compositionality and lexical scoping with incremental program development. This problem is recognized in recent work on separate compilation for Standard ML [AM94, Ler94].

Napier88 [Cut92] is a strongly-typed procedural language with orthogonal persistence and parametric polymorphism. To the best of our knowledge, this is the only environment based on orthogonal persistence that contains a fully integrated compiler.

Galileo [ACO85] is a strongly-typed object-oriented language with a persistent store. Galileo can retain an interactive session (both declarations and executable code) in the persistent store. The store supports strongly-typed locations, so in theory it would be possible to build an integrated environment with Galileo.

Smalltalk [Par90] is a dynamically-typed object-oriented language. It has a programming environment with a persistent store, but the user must explicitly request transfers to and from the store, which strongly limits its applicability. The state of all or part of the system can be saved for later retrieval. The entire system is made available for incremental customization and extension at run-time.

Steps towards persistence exist in some commercial programming environments. The widely-used Macintosh programming environment [App94] provides a rudimentary non-orthogonal persistent store (containing "resource files") that generalizes the file system and a tool (the "resource editor") to modify this store. At run-time, applications can search for and import resources based on their type and identifier.

Existing development tools and CAD tools provide access to a rudimentary persistent store, namely a file system or a database system [KC87, AKMP90]. Software development tools provide revision control for program-related files, which is a rudimentary form of persistence. Commercial CAD tools are built on top of databases and support concurrent engineering. Both sets of tools provide some non-orthogonal persistence.

References

[AB87] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, 1983.

[ACO85] Antonio Albano, Luca Cardelli, and Renzo Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985. Reprinted in [ZM90].

[AGG+94] Joshua S. Auerbach, Arthur P. Goldberg, Germán S. Goldszmidt, Ajei S. Gopal, Mark T. Kennedy, Josyula R. Rao, and James R. Russell. Concert/C: A language for distributed programming, 1994. IBM T. J. Watson Research Center.

[AKMP90] Hamideh Afsarmanesh, David Knapp, Dennis McLeod, and Alice Parker. An extensible object-oriented approach to databases for VLSI/CAD. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, pages 607–618, San Mateo, CA, USA, 1990. Morgan Kaufmann Publishers.

[AM85] Malcolm P. Atkinson and Ronald Morrison. Procedures as persitent data objects. *ACM Transactions on Programming Languages and Systems*, 7(4):539–559, October 1985.

[AM94] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*. ACM, 1994.

[AOC+88] Gregory R. Andrews, Ronald A. Olsson, Michael Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purdin, and Gregg Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[App94] Apple Computer. *Inside Macintosh Library*. Addison-Wesley, 1994.

[AVW93] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang.* Prentice Hall International, 1993.

[BH75] Per Brinch Hansen. The progamming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, 1975.

[BK93] Henri E. Bal and M. Frans Kaashoek. Object distribution in Orca using compile-time and run-time techniques. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93)*, pages 162–177, Washington D.C., USA, September 1993.

[BKLL93] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso. *Programming under Mach.* Addison-Wesley, Reading, MA, USA, 1993.

[BKT92] Henri E. Bal, Frans E. Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[BMO+89] Robert Bretl, David Maier, Allen Otis, Jason Perry, Bruce Schuchardt, Jacob Stein, E. Harold Williams, and Monty Williams. The GemStone data management system. In Won Kim and Frederick H. Lochovsky, editors, *Object Oriented Concepts, Databases and Applications.* Addison-Wesley, Reading, MA, USA, 1989. Chapter 12.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39–59, February 1984.

[BNOW94] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. Research Report 115, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, USA, February 1994.

[BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[Car86] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, Lecture Notes in Computer Science, Volume 242, pages 48–70. Springer-Verlag, Berlin, Germany, 1986.

[Car94] Luca Cardelli. Obliq: A language with distributed scope. Technical report, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, USA, November 1994.

[CC91] Roger S. Chin and Samuel T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[CDG+92] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[CG92] Nicholas Carriero and David Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, February 1992.

[Cla91] Stewart M. Clamen. Data persistence in programming languages: A survey. Technical Report CMU-CS-91-155, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA, May 1991.

[CM82] R. Cole and A. J. Morrison. *An Introduction to Programming with S-Algol.* Cambridge University Press, 1982.

[Cut92] Quintin I. Cutts. *Delivering the Benefits of Persistence to System Construction and Execution.* PhD thesis, University of St Andrews, St Andrews, Fife KY16 9SS, Scotland, 1992.

[CWA+93] Mats Carlsson, Johan Widèn, Johan Andersson, Stefan Andersson, Kent Boortz, Hans Nilsson, and Thomas Sjöland. SICStus Prolog user's manual. Technical report, Swedish Institute of Computer Science, Box 1263, 164 28 Kista, Sweden, 1993.

[Der94] Daniel P. Dern. *The Internet Guide for New Users.* McGraw Hill, 1994.

[DKF94] Dale Dougherty, Richard Koman, and Paul Ferguson. *The Mosaic Handbook for the X Window System.* O'Reilly & Associates, 1994.

[DLAR91] Partha Dasgupta, Richard J. LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The Clouds distributed operating system. *IEEE Computer*, pages 34–44, November 1991.

[DV66] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[DZ83] J. D. Day and H. Zimmermann. The OSI reference model. *Proceedings IEEE*, 71:1334–1340, December 1983.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation Series. The MIT Press, Cambridge, MA, USA, 1994.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* Scientific and Engineering Computation Series. The MIT Press, Cambridge, MA, USA, 1994.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Readings, MA, USA, 1983.

[GS94] Ravi Ganesan and Ravi Sandhu (Guest Editors). Securing Cyberspace. *Communications of the ACM*, 37(11), November 1994. Special Section.

[Hau94] Bogumił Hausman. Turbo Erlang: Approaching the speed of C. In Evan Tick and Giancarlo Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Kluwer Academic, 1994.

[Hew85] Carl Hewitt. The challenge of open systems. *The BYTE Magazine*, 4(10), 1985.

[HI91] Carl Hewitt and Jeff Inman. DAI betwixt and between: From "Intelligent Agents" to Open Systems Science. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1409–1418, November/December 1991.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):47–56, October 1974.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[HSS89] William H. Harrison, John J Shilling, and Peter F. Sweeney. Good news, bad news: Experinece building a software development environment using the object-oriented paradigm. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 85–94. ACM Press, New Orleans, LA, USA, October 1989.

[HSW95] Martin Henz, Gert Smolka, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In V. Saraswat and P. Van Hentenryck, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 27–48. The MIT Press, Cambridge, MA, 1995. To appear.

[HW87] Maurice P. Herlihy and Jeannette M. Wing. Avalon: Language support for reliable distributed systems. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*. IEEE, July 1987.

[HZ87] Mark F. Hornick and Stanley B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transaction on Office Information Systems*, 5(1):70–95, January 1987. Reprinted in [ZM90].

[IBM78] IBM. IBM report on the contents of a sample of programs surveyed, 1978. San Jose, CA, USA.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[JMH93] Sverker Janson, Johan Montelius, and Seif Haridi. Ports for objects. In *Research Directions in Concurrent Object-Oriented Programming.* The MIT Press, Cambridge, MA, USA, 1993.

[KC87] R. Katz and E. Chang. Managing change in a computer-aided design database. In *Proceedings of the XIII International Conference on Very Large Databases*, Brighton, England, 1987. Morgan Kaufmann Publishers.

[KL86] Brian Kantor and Phil Lapsley. Network News transfer protocol. Internet Request for Comments RFC 977, Network Information Center, SRI International, Menlo Park, California, USA, February 1986.

[KMBT92] M. Frans Kaashoek, Raymond Michiels, Henri E. Bal, and Andrew S. Tanenbaum. Transparent fault-tolerance in parallel Orca-programs. In *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, pages 297–312. USENIX, Newport Beach, CA, USA, March 1992.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st Symposium on Principles of Programming Languages*, pages 109–122. ACM, January 1994.

[Lis88] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[LPJ+94] Cricket Liu, Jerry Peek, Russ Jones, Bryan Buus, and Adrian Nye. *Managing Internet Information Services.* O'Reilly & Associates, 1994.

[LT92] Lone Leth and Bent Thomsen. Some Facile chemistry. Technical Report ECRC-92-14, ECRC, European Computer-Industry Research Centre, ECRC GmbH, Arabellastr. 17, D-81925 Munich, May 1992.

[MBC+88] R. Morrison, A. L. Brown, R. Carrick, R. Connor, and A. Dearle. On the integration of object-oriented and process-oriented computation in persistent environments. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, Lecture Notes in Computer Science, Volume 334, pages 334–339. Springer-Verlag, Ebernburg, Germany, September 1988.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall International, New York, USA, 1989.

[Mil92] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[MMS94] Florian Matthes, Sven Müßig, and Joachim W. Schmidt. Persistent polymorphic programming in Tycoon: An introduction. FIDE Technical Report FIDE/94/106, Fachbereich Informatik, Universität Hamburg, Germany, August 1994.

[MT86] Sape J. Mullender and Andrew S. Tanenbaum. The design of a capability-based distributed operating system. *Computer Journal*, 29(4):289–299, April 1986.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML*. The MIT Press, Cambridge, MA, USA, 1990.

[Par90] ParcPlace Systems. Objectworks\Smalltalk. Mountain View, California 94043, 1990.

[Pos82] John Postel. Simple Mail Transfer Protocol. Internet Request for Comments RFC 821, Network Information Center, SRI International, Menlo Park, California, USA, August 1982.

[PR83] J. Postel and J. Reynolds. Telnet protocol specification. Internet Request for Comments RFC 854, Network Information Center, SRI International, Menlo Park, California, USA, May 1983.

[PR85] J. Postel and J. Reynolds. File Transfer Protocol (FTP). Internet Request for Comments RFC 959, Network Information Center, SRI International, Menlo Park, California, USA, October 1985.

[PSWL94] Graham D. Parrington, Santosh K. Shrivastava, Stuart M. Wheater, and Mark C. Little. The design and implementation of Arjuna, October 1994. Deliverable for ESPRIT BRA Project BROADCAST (no 6360).

[RAA+88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, P. Leonard, S. Langlois, and W. Neuhauser. Chorus distributed operating system. *Computing Systems*, 1:305–379, October 1988.

[RC89] Joel E. Richardson and Michael J. Carey. Persistence in the E language: Issues and implementation. *Software - Practice and Experience*, 19(12):1115–1150, December 1989.

[Rei86] S. P. Reiss. An object-oriented framework for graphical programming. *SIGPLAN Notices*, 21(10), October 1986.

[RWW89] William R. Rosenblatt, Jack C. Willeden, and Alexander L. Wolf. Oros: Toward a type model for software development environments. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 297–304. ACM Press, New Orleans, LA, USA, October 1989.

[SDP91] Santosh K. Shrivastava, Graeme N. Dixon, and Graham D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, pages 66–73, January 1991.

[Sha89] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.

[Smo94a] Gert Smolka. The definition of Kernel Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, November 1994.

[Smo94b] Gert Smolka. A foundation for higher-order concurrent constraint programming. In Jean-Pierre Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, pages 50–72, München, Germany, 7–9 September 1994. Springer-Verlag.

[Smo94c] Gert Smolka. An Oz primer. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

[Smo95] Gert Smolka. The definition of Kernel Oz. In Andreas Podelski, editor, *Constraints: Basics and Trends*, Lecture Notes in Computer Science, vol. 910, pages 251–292. Springer-Verlag, 1995. Appears also in the Oz Documentation Series, [Smo94a].

[SS94] Christian Schulte and Gert Smolka. Encapsulated search in higher-order concurrent constraint programming. In Maurice Bruynooghe, editor, *Logic Programming: Proceedings of the 1994 International Symposium*, pages 505–520, Ithaca, New York, USA, 13–17 November 1994. The MIT Press.

[SSW94] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In A.H. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, vol. 874, pages 134–150, Orcas Island, Washington, USA, 2-4 May 1994. Springer-Verlag.

[ST94] Gert Smolka and Ralf Treinen (ed.). DFKI Oz documentation series. Available on paper or via WWW from `http://ps-www.dfki.uni-sb.de/oz`, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D–66123 Saarbrücken, Germany, 1994.

[Ste90a] Guy L. Steele. *Common LISP: The Language*. Digital Press, second edition, 1990.

[Ste90b] W. Richard Stevens. *Unix Network Programming*. Software Series. Prentice Hall International, 1990.

[Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Professional Computing Series. Addison-Wesley, Reading, MA, USA, 1992.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall International, 1995.

[Tic91] Evan Tick. *Parallel Logic Programming*. Series in Logic Programming. The MIT Press, 1991.

[TKB92] Andrew S. Tanenbaum, M. Frans Kaashoek, and Henri E. Bal. Parallel programming using shared objects and broadcasting. *IEEE Computer*, 25(8):10–19, August 1992.

[TLP+93] Bent Thomson, Lone Leth, Sanjiva Prasad, Tsung-Min Kuo, Andre Kramer, Fritz Knabe, and Alessandro Giacalone. Facile Antigua release programming guide. Technical Report ECRC-93-20, ECRC, European Computer-Industry Research Centre, ECRC GmbH, Arabellastr. 17, D-81925 Munich, December 1993.

[Tur93] Louis H. Turcotte. A survey of software environments for exploiting networked computing resources. Technical report, Engineering Center for Computational Field Simulation, P.O. Box 6176, Mississippi State, MS 39762, USA, June 1993.

[TvRvS+90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[Wik94] Claes Wikström. Distributed programming in Erlang. In Hoon Hong, editor, *First International Symposium on Parallel Symbolic Computation (PASCO '94)*, volume 5 of *Lecture Notes Series in Computing*, pages 412–421, Hagenberg/Linz, Austria, September 1994. World Scientific.

[ZM90] Stanley B. Zdonik and David Maier. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1990.