Guardians and Actions: Linguistic Support for Robust, Distributed Programs

Barbara Liskov
Robert Scheifler

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

This paper presents an overview of an integrated programming language and system designed to support the construction and maintenance of distributed programs: programs in which modules reside and execute at communicating, but geographically distinct, nodes. The language is intended to support a class of applications in which the manipulation and preservation of long-lived, on-line, distributed data is important. The language addresses the writing of robust programs that survive hardware failures without loss of distributed information and that provide highly concurrent access to that information while preserving its consistency. Several new linguistic constructs are provided; among them are atomic actions, and modules called guardians that survive node failures.

1. Introduction

Technological advances have made it cost-effective to construct large systems from collections of computers connected via networks. To support such systems, there is a growing need for effective ways to organize and maintain *distributed programs*: programs in which modules reside and execute at communicating, but geographically distinct, locations. In this paper we present an overview of an integrated programming language and system designed for this purpose.

Distributed programs run on *nodes* connected (only) via a communications network. A node consists of one or more processors, one or more levels of memory, and any number of external devices. Different nodes may contain different kinds of processors and devices. The network may be longhaul or shorthaul, or any combination connected by gateways. Neither the network nor any nodes need be reliable. However, we do assume that all failures can be detected as explained in [14]. We also assume that message delay is long relative to the time needed to access local memory, and therefore access to non-local data is significantly more expensive than access to local data.

The applications that can make effective use of a distributed organization differ in their requirements. We have concentrated on a class of applications in which the manipulation and preservation of long-lived, on-line data is important. Examples of such applications are banking systems, airline reservation systems, office automation systems, data base systems, and various components of operating systems. In these systems, real-time constraints are not severe, but reliable, available, distributed data is of primary importance. The systems may serve a geographically distributed organization. Our language is intended to support the implementation and execution of such systems.

The application domain, together with our hardware assumptions, imposes a number of requirements:

Service. A major concern is to provide continuous service of the system as a whole in the face of node and network failures. One principle that applies here is that local problems should be localized. For example, a program should be able to perform its task as long as the particular nodes it needs to communicate with are functioning and reachable. In addition, it should be possible for an application program to use replication of both data and processing as a means for increasing the availability of a service and for providing graceful degradation.

Extensibility. An important reason for wanting a distributed implementation is to make it easy to add and reconfigure hardware in order to increase processing power, decrease response time, or increase the availability of data. Similarly, it should be easy to remove hardware. In the same way that the physical system can be extended or reconfigured, it must be possible to implement logical systems that can be expanded and reconfigured. For example, a banking system might need to grow or shrink to accommodate changing numbers of tellers. To maintain continuity of service, it must be possible to make both logical and physical changes *dynamically*, while the system continues to operate.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661, and in part by the National Science Foundation under grant MCS79-23769.

Autonomy. We assume that nodes are owned by individuals or organizations, and that with ownership comes the desire to control how the node is used. For example, the owner may want control over what runs at the node, or, if the node provides a service to programs running at other nodes, the owner may want control over when that service is available. Further, the node might contain data that must remain resident at that node; for example, a multi-national organization must abide by laws governing information flow among countries. The important point here is that the need for distribution arises not only from efficiency considerations, but from political and sociological considerations as well.

Distribution. The distribution of data and processing can have a major impact on overall efficiency, both in terms of responsiveness, and cost-effective use of hardware. Distribution also affects availability. To create efficient, available systems while retaining autonomy, the programmer needs explicit control over the placement of modules in the system. However, to support a reasonable degree of modularity, changes in location of modules should have limited, localized effects on the actual code.

Concurrency. Another major reason for choosing a distributed implementation is to take advantage of the potential concurrency in an application, thereby increasing efficiency and decreasing response time. For example, the clerks in an airline reservation system should be able to process requests concurrently.

Consistency. In almost any system where on-line data is being read and modified by on-going activities, there are important consistency constraints that must be maintained. For example, in an airline reservation system a flight must not be overbooked by more than a certain number of seats. Such constraints apply not only to individual pieces of data, but to distributed sets of data as well. For example, when funds are transferred from one account to another in a banking system, the net gain over the two accounts must be zero. Also, data that is replicated to increase availability must be kept consistent.

In the remainder of this paper we discuss a programming language and system, called Argus, that satisfies these requirements. To avoid rethinking issues that arise in sequential languages, we have chosen to base Argus on an existing sequential language. CLU [16, 19] was chosen because it supports the construction of well-structured programs through abstraction mechanisms, and because it is an object-oriented language, in which programs are naturally thought of as operating on (potentially) long-lived objects.

Of the above requirements, we found consistency the most difficult to meet. The main issues here are the coordination of concurrent activities (permitting concurrency but avoiding interference), and the masking of hardware failures. Thus, to support consistency we had to devise methods for building a reliable system on unreliable hardware. Reliability is an area that has been almost completely ignored in programming languages (with the exception of [21]). Yet our study of applications convinced us that consistency is a crucial requirement: an adequate language must provide a modular, reasonably automatic method for achieving consistency.

Our approach is to provide *atomicity* as fundamental concepts in the language. The concept of atomicity is not original with our work, having been used extensively in data base applications [3, 4]. However, we believe the integration into a programming language of a general mechanism for achieving atomicity is quite novel. Atomicity is discussed in the next section.

Section 3 presents an overview of Argus. The main features are *guardians*, the logical unit of distribution in our system, and atomic *actions*. Section 4 illustrates most of the important features of the language with a simple mail system. The final section concludes with a discussion of what has been accomplished.

2. Atomicity

Our solution to the problem of maintaining a consistent distributed state in the face of concurrent, potentially interfering activities, and in the face of system failures such as node crashes and network disruptions, is to make activities *atomic*. The distributed state is a collection of data objects that reside at various locations in the network. Some of these objects are *stable*: they are stored on *stable storage devices*, for which the probability of loss of information due to hardware failures is extremely small (see [14]).[1] Other objects are stored in volatile memory. Since the probability of loss of volatile objects is relatively high, these objects must contain only redundant information if the system as a whole is to avoid loss of information. Such redundant information is useful for improving efficiency, e.g., an index for fast access into a data base.

An activity can be thought of as a process that attempts to examine and transform some objects in the distributed state from their current (initial) state to some new (final) state, with any number of intermediate state changes. Two properties distinguish an activity as being atomic: indivisibility and recoverability. By indivisibility, we mean that the execution of one activity never appears to overlap (or contain) the execution of any other activity. If the objects being modified by one activity are observed over time by another activity, the latter activity will either always observe the initial states or always observe the final states, but it will never observe intermediate states. By recoverability, we mean that the overall effect of the activity is all-or-nothing: either all of the objects remain in their initial state, or all change to their final state. If a failure occurs while an activity is running, either it must be possible to complete the activity, or to restore all objects to their initial states.

Supporting these requirements as part of the semantics of a programming language imposes substantial implementation difficulties. However, we believe atomic activities are necessary and are a fairly natural model for a large class of applications. If the language/system does not provide actions, the user will be compelled to implement them, perhaps unwittingly reimplementing them with each new application, and may implement them incorrectly. Therefore, atomicity must be an integral concept of the language.

---

1. We need merely assume that stable storage is accessible to every node in the system; it is not necessary that every node have its own local stable storage devices.

2.1 Actions

We call an atomic activity an *action*. An action may complete either by *committing* or *aborting*. When an action aborts, the effect is as if the action had never begun: all modified objects are restored to their previous state. When an action commits, all modified objects take on their new states; only at this point do changes to stable objects become permanent.

One simple way to implement the indivisibility property is to force actions to run sequentially. However, one of our goals is to provide a system that supports a high degree of concurrency. The usual method of providing indivisibility in the presence of concurrency, and the one we have adopted, is to guarantee *serializability* [5], namely, actions are scheduled in such a way that their overall effect is as if they had been run sequentially in some order. To prevent one action from observing or interfering with the intermediate states of another action, we need to synchronize access to shared objects. In addition, to implement the recoverability property, we need to be able to undo the changes made to objects by aborted actions.

Since synchronization and recovery are likely to be somewhat expensive to implement, we do not provide these properties for all objects. For example, objects that are purely local to a single action do not require these properties. The objects that do provide these properties are called *atomic objects*, and we restrict our notion of atomicity to cover only access to atomic objects. That is, atomicity is guaranteed only when the objects shared by actions are atomic objects.

Atomic objects are encapsulated within *atomic* abstract data types. An abstract data type consists of a set of objects and a set of primitive operations; the primitive operations are the only means of accessing and manipulating the objects [15]. Atomic types have operations just like normal data types, except that operation calls provide indivisibility and recoverability for the calling actions. Some atomic types are built-in while others are user-defined. Argus provides, as built-in types, atomic arrays, records, and variants, with operations nearly identical to the normal arrays, records, and variants provided in CLU. In addition, objects of built-in scalar types, such as characters and integers, are atomic, as are structured objects of built-in *immutable* types, such as strings, whose components cannot change over time.

Our implementation of built-in atomic objects is based on a fairly simple locking model. There are two kinds of locks: read locks and write locks. Before an action uses an object, it must acquire a lock in the appropriate mode. The usual locking rules apply: multiple readers are allowed, but readers exclude writers and a writer excludes readers and all other writers. When a write lock is obtained, a *version* of the object is made, and the action operates on this version. If, ultimately, the action commits, this version will be retained, and the old version discarded. If the action aborts, this version will be discarded, and the old version retained. For example, atomic records have the usual component selection and update operations, but the selection operations obtain a read lock on the record (not the component), and the update operations obtain a write lock and create a version of the record the first time the action modifies the record. Since changes become permanent only at the granularity of entire actions, acquired locks and versions can be kept in volatile storage while an action executes.

All locks acquired by an action are held until the completion of that action, a simplification of standard two-phase locking [7]. This rule avoids the problem of *cascading* aborts: if a lock on an object could be released early, and the action later aborted, any action that had observed the new state of that object would also have to be aborted.

Within the framework of actions, there is a straightforward way to deal with hardware failures at a node: they simply force the node to crash, which in turn forces actions to abort. The volatile information (e.g., versions) kept for an action that has not yet finished will be lost if the node crashes. If this happens the action must be forced to abort. To ensure that the action will abort, a two-phase commit protocol [8] is used. In the first phase, an attempt is made to verify that all locks are still held, and to record the new state of each modified stable object on stable storage. If the first phase is successful, then in the second phase the locks are released, the recorded states become the current states, and the previous states are forgotten. If the first phase fails, the recorded states are forgotten and the action is forced to abort, restoring the objects to their previous states.

Turning hardware failures into aborts has the merit of freeing the programmer from low-level hardware considerations. On the surface it also appears to reduce the probability that actions will commit. However, this is a problem only when the time to complete an action approaches the mean time between failures of the nodes. We believe that most actions are quite short compared to realistic MTBF for hardware available today.

It has been argued that indivisibility is too strong a property for certain applications, because it limits the amount of potential concurrency [13]. We believe that indivisibility is the desired property for most applications, *if* it is required only at the appropriate levels of abstraction. In particular, we intend to provide a mechanism for *user-defined* atomic data types. The important property of these types is that they are free to violate indivisibility internally, but they present an external interface that does not violate indivisibility. We will not present such a mechanism here; the exact linguistic constructs are still a subject of current research.

2.2 Nested Actions

Thus far, we have presented actions as monolithic entities. In fact, it is useful to break down such entities into pieces; to this end we provide hierarchically structured, *nested* actions. Nested actions, or subactions, are a mechanism for coping with failures, as well as for introducing concurrency within an activity. An action may contain any number of subactions, some of which may be performed sequentially, some concurrently. This structure cannot be observed from outside; i.e., the overall action still satisfies the atomicity properties. Subactions appear as atomic activities with respect to other subactions of the same parent. Subactions can commit and abort independently, and a subaction can abort without forcing its parent action to abort. However, the commit of a subaction is conditional: even if all subactions commit, aborting the parent action will abort all of the subactions. Further, changes to stable objects become permanent only when top-level actions commit.

Nested actions aid in composing (and decomposing) activities in a modular fashion. For example, a collection of existing actions can easily be combined into a single, higher-level action, and can be run concurrently within that action with no need for additional synchronization. To extend this example, the concurrent actions might be reads or writes to the sites of a replicated data base. If only a majority of the reads or writes must be successful for the overall action to succeed, this is easily accomplished by committing the overall action once a majority of the subactions commit, even though some of the other subactions aborted.

Nested actions have been proposed by others [4, 22]; our model is similar to that presented in [20]. To keep the locking rules simple, we do not allow a parent action to run concurrently with its children. The rule for read locks is extended so that an action may obtain a read lock on an object provided every action holding a write lock on that object is an ancestor. An action may obtain a write lock on an object provided every action holding a (read or write) lock on that object is an ancestor. When a subaction commits, its locks are inherited by its parent; when a subaction aborts, its locks are discarded.

Note that the locking rules permit multiple writers, which implies that multiple versions of objects are now needed. However, since writers must form a linear chain when ordered by ancestry, and actions cannot execute concurrently with their subactions, only one writer can ever actually be executing at one time. Hence, it suffices to use a stack of versions (rather than a tree) for each atomic object. On commit, the top version becomes the new version for the parent; on abort the top version is simply discarded. A detailed description of locking and version management in a system supporting nested actions is presented in [20].

Since changes become permanent only when top-level actions commit, the two-phase commit protocol is used only for top-level actions. Nested actions do not guarantee any additional reliability in the face of node crashes. Various *checkpoint* mechanisms have been proposed to increase this kind of reliability [9]. Although we are considering ways of including similar mechanisms, it appears they alter fundamentally the semantic view presented to the user, and thus cannot be considered merely as optional features of a system.

2.3 Remote Procedure Call

Perhaps the single most important application of nested actions is in masking communication failures. Logical nodes (described in the next section) in our system communicate via messages. We believe that the most desirable form of communication is the paired send and reply: for every message sent, a reply message is expected. In fact, we believe the form of communication that is needed is *remote procedure call*, with *at-most-once* semantics, namely, that (effectively) either the message is delivered and acted on exactly once, with exactly one reply received, or the message is never delivered and the sender is so informed.

The rationale for the high-level, at-most-once semantics of remote procedure call is presented in [18]. Briefly, we believe the system should mask from the user low-level issues, such as packetization and retransmission, and that the system should make a reasonable attempt to deliver messages. However, we believe the possibility of long delays and of ultimate failure in sending a message cannot and should not be masked. The sender should be allowed to cope with communication failure according to the demands of the particular application, and must be able to terminate communication if the delays become excessive. If communication is terminated, then the remote procedure call should have no effect.

The all-or-nothing nature of remote procedure call is similar to the recoverability property of actions, and the ability to cope with communication delays and failures is similar to the ability of an action to cope with the failures of subactions. Therefore, it seems natural to implement a remote procedure call as a subaction: communication failures will force the subaction to abort, and the sender has the ability to abort the subaction on demand. However, as mentioned above, aborting the subaction does not force the parent action to abort. The sender is free to find some other means of accomplishing its task, such as communicating with some other node.

2.4 Remarks

In our model, there are two kinds of actions: nested actions and top-level actions. We believe these correspond in a natural way to activities in the application system. Top-level actions correspond to activities that interact with the external environment. For example, in an airline reservation system, a top-level action might correspond to an interaction with a clerk who is entering a related sequence of reservations. Nested actions, on the other hand, correspond to internal activities that are intended to be carried out as part of an external interaction; a reservation on a single flight is an example.[2]

Atomic types provide two services to the user of the language: they guarantee indivisibility and recoverability for using actions. The user of our language does not need to write any code to undo or compensate for the effects of aborted actions. On the other hand, the commit of a top-level action is irrevocable. If that action is later found to be in error, actions that compensate for the effects of the erroneous action, and all later actions that depended on it (read its results), must be defined and executed by the user. Note that in general there is no way that such compensation could be done automatically by the system, since extra-system activity is needed (e.g., canceling already issued checks).

Given our use of a locking scheme to implement atomic objects, it is certainly possible for two (or more) actions to *deadlock*, each attempting to acquire a lock held by the other. Although in many cases deadlock can be avoided with careful programming, certain deadlock situations are unavoidable. Our method of breaking deadlocks is to abort actions, rather than refuse locks. Although distributed deadlock detection algorithms that detect a large class of deadlocks are possible (see [20]), the Argus system is not guaranteed to detect deadlocks; in general, deadlocks must be broken by timing out and aborting actions.

3. Linguistic Constructs

In this section we describe the main features of a new language designed to support the requirements discussed in Section 1. The most novel features of this language are the constructs for implementing guardians, the logical nodes of the system, and for implementing actions, as described in the previous section. As stated in the introduction, we have chosen to use the sequential language CLU as a basis for the design. As in CLU, all type-checking in Argus is done at compile time.

---

2. Nested top-level actions are also available. They are useful for accomplishing benevolent side effects, e.g., updating a cache or performing garbage collection or collecting statistics, that need not be undone if the parent aborts.

3.1 Overview

In Argus, a distributed program is composed of a group of *guardians*. A guardian encapsulates one or more resources, and provides controlled access to those resources. The external interface of a guardian consists of a set of operations called *handlers*, which may be invoked by other guardians using the at-most-once, remote procedure call semantics discussed previously. The guardian executes the calls on these handlers, synchronizing them as needed. Furthermore, it may refuse to perform an access desired by a caller if the caller does not have proper authorization.

Internally, a guardian contains data objects and processes. Some of the data objects comprise the global state of the guardian; these objects, such as the actual resources, are shared by the processes. Other objects are local to the individual processes.

Guardians exist entirely at a single physical node: all of a guardian's processes run at that node, and (the volatile state of) the guardian's objects are stored at that node. However, as explained below, a guardian survives crashes of the node at which it resides.

A guardian's global state is a portion of the distributed state and as such may consist of both stable and volatile objects. After a crash of the guardian's node, the language support system re-creates the guardian with the stable objects as they were when last written to stable storage, i.e., as of the last commit of a top-level action that modified some of the guardian's stable objects. A process is started in the guardian to re-create the volatile objects. Once the volatile objects have been restored, the guardian can resume background tasks, and can respond to new requests.

Although the processes inside a guardian can share objects directly, direct sharing of local objects between processes in different guardians is not permitted. The only method of inter-guardian communication is by calling handlers, and the arguments to handlers are passed by value: it is impossible to pass a reference to a local object in a message. This rule ensures that objects local to a guardian remain local, and thus ensures that a guardian retains control of its own objects. It also provides the programmer with a concept of what is expensive: local objects are close by and inexpensive to use, while non-local objects are more expensive to use; this is underlined by the different access methods (procedure call versus handler call). A method for passing data values between heterogeneous nodes using different internal representations is presented in [10].

Guardians and handlers are an abstraction of the underlying hardware of a distributed system. A guardian is a logical node of the system, and inter-guardian communication via handlers is an abstraction of the physical network. While the implementation of a guardian is guaranteed never to be split across physical nodes, for convenience several guardians may reside at the same physical node. Such guardians communicate via handler calls, however: all inter-guardian communication is location-independent. The most important difference between the logical system and the physical system is reliability: the stable state of a guardian is never lost (to a very high probability), and the at-most-once semantics of handler calls ensures that handlers either succeed completely or have no effect.

3.2 Guardian Structure

The syntax of a guardian definition is shown in Figure 1.[3] A guardian definition implements a special kind of abstract data type whose operations are handlers. The name of this type, and the names of the handlers, are listed in the guardian header. In addition, the type provides one or more creation operations, called *creators*, that can be invoked to create new guardians of the type; the names of the creators are also listed in the header. Guardians may be *parameterized*, providing the ability to define a class of related abstractions by means of a single module. Parameterized types are discussed in [16, 19].

Fig. 1. Guardian structure.

```
name = guardian [ [parameter-decls] ]
          is creator-names handles handler-names
   { [ stable ] variable-decls-and-inits }
   [ recover body end ]
   [ background body end ]
   { creator-and-handler-definitions }
   % local procedures and iterators may also be defined
   end name
```

The first internal part of a guardian is a list of variable declarations, with optional initializations defining the guardian state. Some of these variables can be declared as **stable** variables; the others are volatile variables.

The stable state of a guardian consists of all objects *reachable* from the stable variables; these objects, called stable objects, have their new versions written to stable storage by the system when top-level actions commit. Argus, like CLU, has an object oriented semantics. Variables name (or refer to) objects residing in a free storage area. Objects themselves may refer to other objects, permitting recursive and cyclic data structures without the use of explicit pointers. The set of objects reachable from a variable consists of the object that variable refers to, any objects referred to by that object, and so on. (In a language with explicit pointers, the concept of reachability would still be needed to accommodate the use of pointers in stable objects.)

We require that all stable objects also be atomic objects, as discussed in Section 2. This requirement is enforced by compile-time type-checking: the type of each stable variable must be atomic. One reason for this requirement is that the system knows how to synchronize with activity in the guardian to ensure that atomic objects are written to stable storage in internally consistent states. In addition, the system knows how to write atomic objects in an incremental manner and still preserve the sharing among these objects. These same properties do not hold

3. In the syntax, optional clauses are enclosed with [ ], zero or more repetitions are indicated with { }, and alternatives are separated by |. The % sign starts a comment.

for non-atomic objects. As mentioned in Section 2, the language provides a number of built-in atomic types, and users may define new abstract atomic types. In fact, guardians are themselves one class of user-definable atomic types.

Guardian instances are created dynamically by invoking creator operations of the guardian type. For example, suppose we have a guardian definition with header:

g = **guardian is** create **handles** h1, h2, h3

and the *create* operation has header:

create = **creator** (n: int) **returns** (g)

When a process executes

x: g := g$create(3)

the guardian object *x* is created at the same physical node where the process is executing. The handlers provided by the guardian are referred to as *x.h1*, *x.h2* and *x.h3*.

When a creator is invoked, a new guardian instance is created, and any initializations attached to the variable declarations of the guardian state are executed. The body of the creator is then executed; typically, this code will finish initializing the guardian state and then return the guardian object. (Within the guardian, the expression **self** refers to the guardian object.) All three of these steps are performed within a single subaction of the caller: the guardian will be destroyed if the body of the creator aborts this subaction. Aside from creating new guardian instances and executing state variable initializations, creators have essentially the same semantics as handlers, as described further below.

The **recover** section runs after a crash. Before creating a process to run the **recover** section, the system restores the guardian's stable objects from stable storage and executes any initializations attached to declarations of volatile variables of the guardian state. Since updates to stable storage are made only when top-level actions commit, the stable state has the value it had at the latest commit of a top-level action before the guardian crashed. The effects of actions that had executed at the guardian prior to the crash, but had not yet committed to the top level, are lost and the actions are aborted.

The job of the **recover** section is to re-create a volatile state that is consistent with the stable state. This may be trivial, e.g., creating an empty cache, or it might be a lengthy process, e.g., creating a data base index. The **recover** section is not run as an action, although it may create top-level actions, as explained in Section 3.4.[4]

After the successful completion of a creator (when the guardian is first created) or of the **recover** section (after a crash), two things happen inside the guardian: a process is created to run the **background** section, and handler invocations may be executed. The **background** section provides a means of performing periodic (or continuous) tasks within the guardian; an example is presented in Section 4. Like the **recover** section, the **background** section is not run as an action.

3.3 Handlers

Handlers (and creators), like procedures in CLU, are based on the termination model of exception handling [17]. A handler can terminate in one of a number of conditions: one of these is considered to be the "normal" condition, while others are "exceptional," and are given user-defined names. Results can be returned both in the normal and exceptional cases; the number and types of results can differ among conditions. The header of a handler definition lists the names of all exceptional conditions and defines the number and types of results in all cases. For example,

file_date = **handler** (fn: file_name) **returns** (date)
signals (not_possible(string))

is the header of a handler whose calls either terminate normally, returning a result of type date, or exceptionally in condition *not_possible* with a string result. In addition to the named conditions, any handler can terminate in the *failure* condition, returning a string result; failure termination may be caused explicitly by the user code, or implicitly by the system when something unusual happens, as explained further below.

Handler calls differ from ordinary procedure calls in several important ways:

1. Procedures always run inside the guardian in which they are called. Handlers usually belong to some other guardian (although a call to a handler of your own guardian is permitted), and that guardian is likely to reside on some other node. Thus, the system will construct a message containing the arguments and send it to the appropriate node. When the handler call terminates, the system constructs another message containing the termination condition and results, and sends it back to the calling guardian.[5]

2. Procedure arguments and results are passed *by sharing* (see [19]); i.e., the argument and result objects are shared between the calling and called procedure. As mentioned above, handler arguments and results are always passed by value.

3. To achieve the at-most-once semantics discussed previously, handlers are executed as subactions of the calling action. Procedures simply execute within the calling action.

Since a handler executes as an action, it must, in addition to returning or signalling, either commit or abort. We expect committing to be the most common case, and therefore execution of a **return** or **signal** statement indicates commitment. To cause an abort, the **return** or **signal** is prefixed with **abort**.

Let us examine a step-by-step description of what the system does when a handler is invoked:

4. A process that is not running as an action is severely restricted in what it can do. For example, it cannot call operations on atomic objects without first creating a top-level action.

5. If the calling and called guardians reside on the same node, the system may be able to avoid this message passing.

1. A new subaction is created.

2. A message containing the arguments is constructed. Since part of building this message involves executing user-defined code (see [10]), message construction may fail. If so, the subaction aborts and the call terminates with a *failure* exception.

3. The system suspends the calling process and sends the message to the target guardian. If the handler's guardian no longer exists, the subaction aborts and the call terminates with a *failure* exception.

4. The system makes a reasonable attempt to deliver the message, but success is not guaranteed. The reason is that it may not be sensible to guarantee success under certain conditions, such as a crash of the target node. In such cases, the subaction aborts and the call terminates with a *failure* exception. The meaning of such a failure is that there is very low probability of the call succeeding if it is repeated immediately.

5. The system creates a process at the receiving guardian to execute the handler. Note that multiple instances of the same handler may execute simultaneously. The system takes care of locks and versions of atomic objects used by the handler in the proper manner, according to whether the handler commits or aborts. When the handler terminates, the system destroys the process.

6. The system creates the response message and sends it to the calling guardian. If this is impossible (as in (2) or (4) above), the subaction aborts and the call terminates with a *failure* exception.

7. The calling process continues execution. Its control flow is affected by the termination condition as explained in [17]. For example, for a call of file_date above we might have

```
d: date := file_date(fn)    % normal return
   except when not_possible, failure (why: string):
                            ... % exceptional return
          end
```

As was mentioned above (in step 4), the system does not guarantee to deliver messages; it merely guarantees that if message delivery fails there is a very low probability of the call succeeding if it is repeated immediately. Hence, there is no reason for user code to repeatedly retry handler calls. Rather, user programs should guarantee progress by retrying top-level actions, which may fail because of node crashes even if all handler calls succeed.

3.4 Inline Actions

The preceding section explained handler calls only in terms of subactions. Top-level actions are created by means of the action statement:

enter topaction *body* **end**

This causes the *body* to execute as a new top-level action. When the *body* completes, it does so either by committing or aborting. It is also possible to have an inline subaction:

enter action *body* **end**

This causes the *body* to run as a subaction of the action that executes the **enter**.

When an inline action terminates, it must indicate whether it is committing or aborting. Since committing is assumed to be most common, it is the default; the qualifier **abort** can be prefixed to any termination statement to override this default. For example, an inline action can execute

leave

to commit and cause execution to continue with the statement following the **enter** statement; to abort and have the same effect on control, it executes

abort leave

Falling off the end of the *body* causes the action to commit.

Examples of inline actions are given in Section 4.

3.5 Concurrency

The language as defined so far allows concurrency between actions, but not within a single action. To allow subactions to run concurrently, we provide the following statement form:

coenter { *coarm* }+ **end**

where

coarm ::= *armtag* [**foreach** *decl-list* **in** *iter-invocation*]
body

armtag ::= **action** | **topaction**

The process executing the coenter, and the action (if any) on whose behalf it is executing, are suspended; they resume execution after the coenter is finished.

A **foreach** clause indicates that multiple instances of the coarm will be activated, one for each item (a collection of objects) yielded by the given iterator invocation.[6] Each such coarm will have local instances of the variables declared in the *decl-list*, and the objects constituting the yielded item will be assigned to them. Execution of the **coenter** starts by running each of the iterators to completion, sequentially, in textual order. Then all coarms are started simultaneously as concurrent siblings. Each coarm instance runs in a separate process, and each process executes within a new top-level action or subaction, as specified.

A simple example making use of **foreach** is

```
coenter action foreach i: int in int$from_to (1, 5)
  p(i)
  end
```

which creates five processes, each with a local variable *i*, having the value 1 in the first process, 2 in the second process, and so on. Each process runs in a newly created subaction.

A coarm may terminate without terminating the entire **coenter** either by falling off the end of its *body*, or by executing a **leave** statement. As before, **leave** may be prefixed by **abort** to cause the completing action to abort; otherwise the action commits.

A coarm also may terminate by transferring control outside the **coenter** statement. Before such a transfer can occur, all

---

6. An iterator is a limited kind of coroutine that provides results to its caller one at a time [16, 19].

other active coarms of the **coenter** must be terminated. To accomplish this, the system forces all coarms that are not yet completed to abort. To abort a coarm, the system waits for its process to leave any critical regions (see next section); it then destroys the process and aborts the action.

A simple example where such early termination is useful is in timing out a handler call:

```
coenter
  action  x.h(...); exit done
  action  sleep(units); exit timed_out
  end
```

Whichever of these two actions completes first, it commits itself and aborts the other. In either case, the abort takes place immediately (since there are no critical regions). In particular, it is not necessary for the handler call, *x.h(...)*, to finish before the calling action can be aborted. This last fact is important, since the reason for timing out a call may be to avoid waiting a long time due to crashes, loops, or deadlocks elsewhere in the system. (Such timeouts can result in *orphan* processes that continue to run at the called guardian and elsewhere. We have developed algorithms for dealing with orphans, but they are beyond the scope of this paper.)

There is another form of **coenter** for use outside of actions, as in the **recover** and **background** sections of a guardian. In this form the *armtag* is **process**. The semantics is as above, except that no actions are created.

3.6 Synchronization

By now we have lots of potential concurrency. The **background** code of a guardian may have many concurrent tasks in progress. Multiple handler calls may be active inside the guardian, and some of these may have created concurrent subactions. How is all this concurrent activity synchronized?

We expect that most concurrent activity within the action system will be synchronized automatically through the use of atomic objects. For example, consider a guardian that guards a single atomic array and provides a number of handlers, some of which read the array while others modify it. These handlers will be synchronized by their use of the atomic array. There might be several handler calls concurrently reading the array, while handler calls wanting to modify the array will wait for the system to release the locks held by the actions on whose behalf reading is taking place.

However, not all execution takes place within the action system. The **background** and **recover** sections are not actions, and may use the **coenter** statement to perform tasks with concurrent processes. Some mechanism is need to synchronize such processes. In addition, a process synchronization mechanism is needed when implementing highly concurrent user-defined atomic types, and also occasionally to schedule handler calls. To support both needs, we provide a form of *critical region* by means of a built-in type called *mutex*. No two processes may execute simultaneously in critical regions controlled by the same mutex object. Each mutex object has a data object associated with it. We guarantee that, while a process executes in a critical region controlled by the mutex object, the system is prevented from writing the associated data object to stable storage. This latter guarantee can be used to ensure that only consistent states of the associated data object are written to stable storage.

The main interaction of critical regions with the material presented in previous sections is that when a coarm transfers control outside its **coenter** statement, the other processes inside the **coenter** must be terminated and their associated actions, if any, aborted. This termination happens immediately unless a process is in a critical region; in this case, the system allows the process to continue until it exits all critical regions. The assumption is that processes communicate only via atomic data, or via data protected by critical regions. The above rule ensures that a process will be terminated as quickly as possible, but not while the data it shares with other processes is in an inconsistent state.

3.7 Remarks

The language sketched above has two main concepts: guardians and actions. Guardians maintain complete local control over their local data. The data inside a guardian is truly local; no other guardian has the ability to access or manipulate the data directly. The guardian provides access to the data via handler calls, but the actual access is performed inside the guardian. It is the guardian's job to guard its data in three ways: by synchronizing concurrent access to the data, by requiring that the caller of a handler have the authorization needed to do the access, and by making enough of the data stable so that the guardian as a whole can survive crashes without loss of information.

While guardians are the unit of modularity, actions are the means by which distributed computation takes place. A top-level action will start at some guardian. This action can perform a distributed computation by making handler calls to other guardians; those handler calls can make calls to still more guardians, and so on. Since the entire computation is an atomic action, it is guaranteed that the computation is based on a consistent distributed state, and that when the computation finishes, the state is still consistent, assuming in both cases that user programs are correct.

To provide this guarantee, the system must do a lot of work. It keeps track of the history of actions: which guardians are visited, which objects are read, and which are modified. As subactions commit and abort, this history is modified appropriately. Finally, when a top-level action commits, this history is used to ensure that none of the guardians involved[7] have crashed since they were used. If this condition is met, the system updates stable storage appropriately, releases locks, and discards old versions. If the condition is not met, the system forces the action to abort, releases all locks, and restores old versions.

4. A Simple Mail System

In this section we present a very simple mail system. We have designed the system somewhat along the lines of Grapevine [1]. Although we have chosen inefficient implementations for some features, and have omitted many necessary and desirable features of a real mail system, we hope to give some idea of how a real system could be implemented in Argus.

The interface to the mail system is quite simple. Every user has a unique name (*user_id*) and a mailbox. However, mailbox

---

7. The guardians involved are those visited by handler calls performed as subactions of the top-level action, where the subaction and all of its ancestors have committed.

locations are completely hidden from the user. Mail can be sent to a user by presenting the mail system with the user's *user_id* and a *message*; the message will be appended to the user's mailbox. Mail can be read by presenting the mail system with a user's *user_id*; all messages are removed from the user's mailbox and are returned to the caller. For simplicity, there is no protection on this operation: any user may read another user's mail. Finally, there is an operation for adding new users to the system, and some operations for dynamically extending the mail system.

All operations are performed within the action system. For example, a message is not really added to a mailbox unless the sending action commits, messages are not really deleted unless the reading action commits, and a user is not really added unless the requesting action commits.

The mail system is implemented out of three kinds of guardians: *mailers, maildrops* and *registries*. Mailers act as the front end of the mail system: all use of the system occurs through calls of mailer handlers. To achieve high availability, many mailers will exist, e.g., one at each physical node. A maildrop contains the mailboxes for some subset of users. Individual mailboxes are not replicated, but multiple, distributed maildrops are used to reduce contention and to increase availability, in that the crash of one physical node will not make all mailboxes unavailable. The mapping from user_id to maildrop is provided by the registries. Replicated registries are used to increase availability, in that at most one registry need be accessible to send or read mail. Each registry contains the complete mapping for all users. In addition registries keep track of all other registries.

Figure 2 defines a number of abbreviations for atomic types used in implementing the mail system. For simplicity, we use only types obtained from the built-in atomic type generators *struct* and *atomic_array*, together with the abstract types *user_id* and *message*, whose implementations we omit. Structs are *immutable* records: new components cannot be stored in a struct object once it is built. Since structs are immutable, they are atomic. Atomic arrays are one-dimensional, and can grow and shrink dynamically. Of the array operations used in the mail system, *new* creates an empty array, *addh* adds an element to the high end, *trim* removes elements, *elements* iterates over the elements from low to high, and *copy* copies an array. Read locks on the entire array are obtained by *new*, *elements*, and *copy*, and write locks are obtained by *addh* and *trim*.

The mailer guardian definition is presented in Figure 3. A mailer must be given a registry when created; this registry is the mailer's stable "handle" on the entire mail system. The mailer also keeps a volatile handle: the registry representing the "best" access path into the system. The **background** code is used to periodically choose a new registry to play this role; the closest responding registry would be an appropriate choice.

A mailer performs a request to send or read mail by first using the *best* registry to determine the maildrop of the specified

Fig. 2. Abbreviations

```
mailbox      = struct[mail: messagelist,      % messages for
                      user: user_id]          % this user
messagelist  = atomic_array[message]
mailboxlist  = atomic_array[mailbox]
registrylist = atomic_array[registry]
steeringlist = atomic_array[steering]
steering     = struct[users: userlist,        % users with mailboxes
                      drop:  maildrop]        % at this maildrop
userlist     = atomic_array[user_id]
```

Fig. 3. Mailer Guardian

```
mailer = guardian is create
                handles send_mail, read_mail,
                        add_user, add_maildrop, add_registry

stable some: registry   % stable handle
best: registry          % volatile handle

recover
   best := some         % reassign after crash
   end

background
   while true do
      enter topaction
         best := ... % choose closest responding registry
         end
      sleep(...)
      end
   end

create = creator (reg: registry) returns (mailer)
   some := reg
   best := reg
   return(self)
   end create

send_mail = handler (user: user_id, msg: message)
                                        signals (no_such_user)
   best.lookup(user).send_mail(user, msg)
      resignal no_such_user
   end send_mail

read_mail = handler (user: user_id) returns (messagelist)
                                    signals (no_such_user)
   return(best.lookup(user).read_mail(user))
      resignal no_such_user
   end read_mail

add_user = handler (user: user_id) signals (user_exists)
   drop: maildrop := best.choose()
   all: registrylist := best.all_registries()
   coenter
      action
         drop.add_user(user)
      action foreach reg: registry in registrylist$elements(all)
         reg.add_user(user, drop)
            abort resignal user_exists
      end
   end add_user


add_maildrop = handler ()
   all: registrylist := best.all_registries()
   drop: maildrop := maildrop$create()
   coenter action foreach reg: registry in registrylist$elements(all)
      reg.add_maildrop(drop)
      end
   end add_maildrop.

add_registry = handler ()
   all: registrylist := best.all_registries()
   new: registry := registry$create(all, best.all_steerings())
   coenter action foreach reg: registry in registrylist$elements(all)
      reg.add_registry(new)
      end
   end add_registry

end mailer
```

user, and then forwarding the request to that maildrop. A mailer adds a new user by first using the *best* registry to choose a maildrop, and then concurrently asking that maildrop to create a mailbox and informing all registries of the new user/maildrop pair. Note that if the user is discovered to exist at any registry, the overall action aborts.

A new registry is added by extracting the entire user-to-maildrop mapping and the list of all registries from the *best* registry, and using them to create a new registry. The other registries are then informed of the new registry so they may add it to their registry lists. Finally, a new maildrop is added by creating one and informing all registries of its existence.

Figure 4 shows an implementation of the registry guardian. The state of a registry consists of an array of registries, together with a *steering list* associating an array of users with each maildrop. When a registry is created, it is given an array of all other registries, to which it adds itself, and the current steering list. The *add_user* handler checks to make sure the user is not already present, and adds the user to the user array for the given maildrop. The *add_maildrop* and *add_registry* handlers perform no error-checking because correctness is guaranteed by the mailer guardian.

An implementation of the maildrop guardian is given in Figure 5. The state of a maildrop consists of an array of mailboxes; a mailbox is represented by a struct containing a user_id and an array of messages. A maildrop is created with no mailboxes. The *add_user* handler can be invoked to add a mailbox. Note that this handler does not check to see if the user already exists; this checking is performed by the registries. The *send_mail* and *read_mail* handlers use linear search to find the correct mailbox. When the mailbox is found, *send_mail* appends a message to the end of the message array; *read_mail* first copies the array, then deletes all messages, and finally returns the copy. Both handlers assume the user exists; this is guaranteed by the registries.

Finally, in Figure 6, we show a simple use of the mail system, namely, sending a message to a list of users, with the desire that the message be delivered only if all of the users exist, and otherwise to get back a list of all non-existent users. The message is sent to all of the users simultaneously, and the non-existent users are collected in an array. Although a non-atomic array is used, its *addh* operation is defined to be indivisible, so no explicit synchronization is needed here. After all sends are completed, if the array is non-empty, the overall action is aborted, thus ensuring that none of the users are sent mail.

4.1 Remarks

Close examination of the mail system will reveal many places where the particular choice of data representation leads to far less concurrency than might be expected. For example, in the maildrop guardian, since both *send_mail* and *read_mail* modify the message array in a mailbox, either operation will lock out all other operations on the same mailbox until the executing action commits to the top level. Even worse, since both *send_mail* and *read_mail* read the mailbox array, and *add_user* modifies that array, an *add_user* operation will lock out all operations on all mailboxes at that maildrop. In the registry guardian, an *add_user* operation will lock out *lookup* operations on all users with mailboxes at the given maildrop, and an *add_maildrop* operation locks out all *lookup* operations.

In a real system, this lack of concurrency would probably be unacceptable. What is needed are data types that allow more concurrency than simple atomic arrays. For example, an

Fig. 4. Registry Guardian

```
registry = guardian is create
                    handles lookup, choose, all_registries, all_steerings,
                            add_user, add_maildrop, add_registry

stable registries: registrylist   % all registries
stable steerings: steeringlist    % all users and maildrops

create = creator (rest: registrylist, steers: steeringlist) returns (registry)
    registrylist$addh(rest, self)  % add self to list
    registries := rest
    steerings := steers
    return(self)
    end create

lookup = handler (user: user_id) returns (maildrop)
                                 signals (no_such_user)
    for steer: steering in steeringlist$elements(steerings) do
        for usr: user_id in userlist$elements(steer.users) do
            if usr = user
              then return(steer.drop) end
            end
        end
    signal no_such_user
    end lookup

choose = handler () returns (maildrop) signals (none)
    if steeringlist$empty(steerings)
      then signal none end
    drop: maildrop := ... % choose, e.g., maildrop with least users
    return(drop)
    end choose

all_registries = handler () returns (registrylist)
    return(registries)
    end all_registries

all_steerings = handler () returns (steeringlist)
    return(steerings)
    end all_steerings

add_user = handler (user: user_id, drop: maildrop) signals (user_exists)
    for steer: steering in steeringlist$elements(steerings) do
        for usr: user_id in userlist$elements(steer.users) do
            if usr = user
              then signal user_exists end
            end
        if steer.drop = drop
          then userlist$addh(steer.users, user) end   % append user
        end
    end add_user

add_maildrop = handler (drop: maildrop)
    steeringlist$addh(steerings, steering${users: userlist$new(),
                                           drop:  drop})
    end add_maildrop

add_registry = handler (reg: registry)
    registrylist$addh(registries, reg)
    end add_registry

end registry
```

Fig. 5. Maildrop Guardian

```
maildrop = guardian is create handles send_mail, read_mail, add_user

stable boxes: mailboxlist := mailboxlist$new()

create = creator() returns (maildrop)
   return(self)
   end create

send_mail = handler (user: user_id, msg: message)
   for box: mailbox in mailboxlist$elements(boxes) do
      if box.user = user
        then messagelist$addh(box.mail, msg)  % append message
             return
        end
      end
   end send_mail

read_mail = handler (user: user_id) returns (messagelist)
   for box: mailbox in mailboxlist$elements(boxes) do
      if box.user = user
        then mail: messagelist := messagelist$copy(box.mail)
             messagelist$trim(box.mail, 1, 0)  % delete messages
             return(mail)
        end
      end
   end read_mail

add_user = handler (user: user_id)
   mailboxlist$addh(boxes, mailbox${mail: messagelist$new(),
                                    user: user})
   end add_user

end maildrop
```

Fig. 6. Distributing Mail

```
distribute_mail = proc (m: mailer, users: idlist, msg: message)
                                  signals (no_such_users(idlist))
   idlist = array[user_id]
   enter action
      bad: idlist := idlist$new()
      coenter action foreach user: user_id in idlist$elements(users)
         m.send_mail(user, msg)
           except when no_such_user:
                       idlist$addh(bad, user)  % indivisible
                  end
         end
      if ~idlist$empty(bad)
        then abort signal no_such_users(bad) end
      end
   end distribute_mail
```

associative memory that allowed concurrent insertions and lookups could replace the mailbox array in maildrops and the steering list in registries; a queue with a "first-commit first-out" semantics, rather than a "first-in first-out" semantics, could replace the message arrays in maildrops. Such types can be built as user-defined atomic types, although we will not present implementations here.

The concurrency that *is* built in to the mail system leads to a number of potential deadlock situations. For example, in the registry guardian, two instances of *add_user* could simultaneously read the same user array, and then simultaneously attempt to modify that array, neither succeeding because the other still holds a read lock. In the mailer guardian, deadlock is possible if two different *add_user*, *add_maildrop*, or *add_registry* requests modify registries in opposite orders.

Some of these deadlock situations would go away if data representations allowing more concurrency were used. For example, the use of a highly concurrent associative memory for the steering list would allow *add_maildrop* requests to run concurrently. In other cases, the algorithms must be modified. For example, to avoid a deadlock between two different requests to add the same user, the mailer *add_user* operation could pick a distinguished registry, such as the first one in the list of all registries, and perform the registry *add_user* operation there sequentially before performing all of the rest concurrently. To avoid deadlock between concurrent *add_maildrop* and *add_registry* requests, the mailer *add_registry* operation could first get a write lock on the registry list of a distinguished registry, and *add_maildrop* could be forced to obtain its registry list from that same registry.

It may be argued that the strict serialization of actions enforced by the particular implementation we have shown is not important in a real mail system. This does not mean that actions are inappropriate in a mail system, just that the particular granularity of actions we have chosen may not be the best. For example, if an action discovers that a user does (or does not) exist, it may not be important that the user continues to exist (or not exist) for the remainder of the overall action. It is possible to build such "loopholes" through appropriately defined abstract types. As another example, it might not be important for all registries to have the most up-to-date information, provided they receive all updates eventually. In particular, when adding a new user, it may suffice to guarantee that all registries eventually will be informed of that user. This could be accomplished by keeping appropriate information in the stable state of a mailer guardian, and having the background process of that mailer be responsible for eventually informing all registries.

5. Summary and Conclusions

In this paper we have presented a fairly high-level overview of a new language for writing distributed programs. Although a great many details have been omitted, we believe enough of the language has been described to indicate how the requirements stated in the introduction have been met:

Consistency. Actions and atomic objects provide a powerful, easy to use mechanism for ensuring consistency of distributed information. Guardians ensure that the effects of completed activities are not lost in node crashes.

Service. Service is a basic aspect of the language, in that each activity uses just those guardians of interest. Of course, the underlying system components managing those guardians must be highly available, but these are local to the physical nodes executing those guardians. Replication of both data and processing at the application program is achieved through the use of multiple guardians, as seen in the simple mail system presented above.

Distribution. Guardians give the application program control of distribution. Tightly coupled processing and data can be grouped together within a single guardian, which allows for fast local processing. Furthermore, the application program specifies where guardians reside.

Concurrency. A great deal of concurrency is possible between actions, as well as within actions, through the use of concurrent subactions. However, user-defined atomic types may be required to achieve acceptable degrees of concurrency, as illustrated in the mail system.

Extensibility. Guardians can be created dynamically; they can also be destroyed, and moved from one physical node to another, although we have not discussed these latter capabilities. Thus an application program can be reconfigured as needed. Reconfiguration has a minimal impact on guardians that use the application because communication between guardians is location-independent. The syntax of handler calls is location-independent, ensuring that distinct guardians that are instances of the same guardian definition can be used interchangeably. In addition, the names of handlers are location-independent, permitting a guardian to be moved from one node to another without affecting the users of that guardian. Examples of expanding an existing service dynamically can be seen in the mail system.

Autonomy. Guardians have complete control of their local data and resources, and the application program controls where guardians reside; the system never moves guardians on its own initiative. Guardians can be freely created at the same node as the creating guardian, but to create a guardian at a foreign node (or to move a guardian to a foreign node) an intermediary guardian must be used; this guardian can check for proper authorization. However, the owner of a node may wish to allow a particular guardian to be created at that node but disallow that guardian from creating other guardians at the node. Thus our protection model is inadequate, and we are investigating better models.

Argus is quite different from other languages that address concurrent or distributed programs (e g., [2, 6, 11, 12]). These languages tend to provide modules with a superficial resemblance to guardians, and some form of communication between modules based on message passing. For the most part, however, the modules have no internal concurrency and contain no provision for long-term storage that survives crashes. Indeed, many such languages completely ignore the problem of node crashes. In the area of communication, either a low-level, unreliable mechanism is provided, or reliability is ignored, implying that the mechanism is completely reliable, with no way of actually achieving such reliability.

We have completed a preliminary, centralized, partially simulated implementation of the language, ignoring hard problems such as flow control, deadlock detection, and orphan detection. We expect to begin work on a real, distributed implementation in 1982. At this point it is unclear how efficient such an implementation can be. As we have argued, actions are necessary for many applications, so they must be implemented; we expect the implementation will be more efficient at system level than at the application level. For other applications, actions may simply be a convenient tool, not a strictly necessary one. We conjecture that actions can be implemented efficiently enough that they will be used in many applications even when they are not strictly necessary.

Using our initial implementation as a test bed, we have worked out several distributed programs, some abstracted from the applications of interest, others from within the system itself. We have been pleased with the language so far. For example, actions are a useful tool in thinking about the interface to an application. However, we expect to get a much more realistic idea of the strengths and weaknesses of the language when the distributed implementation is complete and we can run real applications. We expect to go through another language/system design cycle after we have gained some experience with such applications.

References

1. Birrell, A., Levin, R., and Schroeder, M., "Grapevine", Xerox PARC, Palo Alto, CA, April 1981. To appear in Communications ACM.

2. Brinch Hansen, P., "Distributed processes: a concurrent programming concept", *Communications ACM 21, 11*, November 1978, 934-941.

3. Davies, C.T., "Recovery semantics for a DB/DC system", *Proceedings of the 1973 ACM National Conference*, 1973, 136-141.

4. Davies, C.T., "Data processing spheres of control", *IBM Systems Journal 17, 2*, 1978, 179-198.

5. Eswaren, K.P, Gray, J.N, Lorie, R.A., and Traiger, I.L., "The notion of consistency and predicate locks in a database system", *Communications ACM 19, 11*, November 1976, 624-633.

6. Feldman, J.A., "High Level Programming for Distributed Computing", *Communications ACM 22, 6*, June 1979, 353-368.

7. Gray, J.N., Lorie, R.A., Putzolu, G.F., and Traiger, I.L, "Granularity of locks and degrees of consistency in a shared data base", *Modeling in Data Base Management Systems*, G.M. Nijssen editor, North Holland, 1976.

8. Gray, J.N., "Notes on data base operating systems", *Lecture Notes in Computer Science 60*, Goos and Hartmanis editors, Springer-Verlag, Berlin, 1978, 393-481.

9. Gray, J.N., et al. "The recovery manager of a data management system", *IBM Research Report RJ2623*, August 1979.

10. Herlihy, M. and Liskov, B., "A value transmission method for abstract data types", *Computation Structures Group Memo 200-1*, MIT Laboratory for Computer Science, Cambridge, MA, July 1981, submitted to ACM TOPLAS.

11. Hoare, C.A.R., "Communicating sequential processes", *Communications ACM 21, 8*, August 1978, 666-677.

12. Ichbiah, J.D. et al., "Preliminary ADA reference manual", *SIGPLAN Notices 14, 6*, June 1979.

13. Lamport, L., "Towards a theory of correctness for multi-user data base systems", *Report CA-7610-0712*, Massachusetts Computer Associates, Wakefield, MA, October 1976.

14. Lampson, B. and Sturgis, H. "Crash recovery in a distributed data storage system", Xerox PARC, Palo Alto, CA, April 1979.

15. Liskov, B. and Zilles, S. N., "Programming with abstract data types", Proceedings ACM SIGPLAN Conference on Very High Level Languages, *SIGPLAN Notices 9, 4*, April 1974, 50-59.

16. Liskov, B., Snyder, A., Atkinson, R.R., and Schaffert, J.C. "Abstraction mechanisms in CLU", *Communications ACM 20, 8*, August 1977, 564-576.

17. Liskov, B. and Snyder, A., "Exception handling in CLU", *IEEE Transactions on Software Engineering 5, 6*, November 1979, 546-558.

18. Liskov, B., "On linguistic support for distributed programs", *Proceedings, IEEE Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, PA, July 1981, 53-60.

19. Liskov, B. et al., "CLU reference manual", *Lecture Notes in Computer Science 114*, Goos and Hartmanis editors, Springer-Verlag, Berlin, 1981.

20. Moss, J.E.B., "Nested transactions: an approach to reliable distributed computing", Ph.D thesis, *Technical Report MIT/LCS/TR-260*, MIT Laboratory for Computer Science, Cambridge, MA, 1981.

21. Randell, B. "System structure for software fault tolerance", *IEEE Transactions on Software Engineering 1, 2*, June 1975, 220-232.

22. Reed, D.P., "Naming and synchronization in a decentralized computer system", Ph.D thesis, *Technical Report MIT/LCS/TR-205*, MIT Laboratory for Computer Science, Cambridge, MA, 1978.