Guardians and Actions: Linguistic Support for Robust, Distributed Programs

BARBARA LISKOV and ROBERT SCHEIFLER
Massachusetts Institute of Technology

An overview is presented 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 concerned with the manipulation and preservation of long-lived, on-line, distributed data. 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.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; *distributed databases*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types*; *concurrent programming structures*; *modules, packages*; D.4.5 [**Operating Systems**]: Reliability—*checkpoint/restart*; *fault-tolerance*; H.2.4 [**Database Management**]: Systems—*distributed systems*; *transaction processing*

General Terms: Languages, Reliability

Additional Key Words and Phrases: Atomicity, nested atomic actions, remote procedure call

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, called ARGUS, that was designed for this purpose.

A preliminary version of this paper appeared in the Conference Record of the Ninth Annual Symposium on Principles of Programming Languages, January 1982 [18].

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 MCS 79-23769.

Authors' address: Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

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 long haul or short haul, 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 [15]. We also assume that message delay is long relative to the time needed to access local memory and therefore that access to nonlocal 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 concerned with the manipulation and preservation of long-lived, on-line data. Examples of such applications are banking systems, airline reservation systems, office automation systems, database 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 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. Failures should be localized so that a program can perform its task as long as the particular nodes it needs to communicate with are functioning and reachable. Adherence to this principle permits an application program to use replication of data and processing to increase availability.

Reconfiguration. An important reason for wanting a distributed implementation is to make it easy to add and reconfigure hardware to increase processing power, decrease response time, or increase the availability of data. It also must be possible to implement logical systems that can be reconfigured. To maintain continuous service, it must be possible to make both logical and physical changes *dynamically*, while the system continues to operate.

Autonomy. We assume that nodes are owned by individuals or organizations that want to control how the node is used. For example, the owner may want to control what runs at the node, or to control the availability of services provided at the node. Further, a node might contain data that must remain resident at that node; for example, a multinational 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, in terms of both 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 the 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.

Consistency. In almost any system where on-line data is being read and modified by ongoing activities, there are consistency constraints that must be maintained. 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.

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 [22, 25, 28]). 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 a fundamental concept in the language. The concept of atomicity is not original with our work, having been used extensively in database applications [4–6, 8–10]. However, we believe the integration into a programming language of a general mechanism for achieving atomicity is novel.

The remainder of the paper is organized as follows. 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 many features of the language with a simple mail system. The final section discusses what has been accomplished.

2. ATOMICITY

Data consistency requires, first of all, that the data in question be resilient to hardware failures, so that a crash of a node or storage device does not cause the loss of vital information. Resiliency is accomplished by means of redundancy. We believe the most practical technique using current technology is to keep data on stable storage devices [15].[1] Of course, stable storage, in common with any other technique for providing resiliency, cannot guarantee that data survive all failures, but it can guarantee survival with extremely high probability.

Data resiliency only ensures data survival in a quiescent environment. Our solution to the problem of maintaining consistent distributed data in the face of concurrent, potentially interfering activities, and in the face of system failures such as node crashes and network disruptions while these activities are running, is to make activities *atomic*.

The state of a distributed system is a collection of data objects that reside at various locations in the network. An activity can be thought of as a process that

[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.

attempts to examine and transform some objects in the distributed state from their current (initial) states to new (final) states, 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. 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, it must be possible either to complete the activity or to restore all objects to their initial states.

2.1 Actions

We call an atomic activity an *action*. An action may complete either by *committing* or by *aborting*. When an action aborts, the effect is as if the action had never begun: all modified objects are restored to their previous states. When an action commits, all modified objects take on their new states.

One simple way to implement the indivisibility property is to force actions to run sequentially. However, one of our goals is to provide 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* [6]; 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 [21]. Atomic types have operations just like normal data types, except that the operations 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 those on the normal arrays, records, and variants provided in CLU [20]. 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 (mutable) 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 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 selection operations obtain a read lock on the record (not the component), and update operations obtain a write lock and create a version of the record the first time the action modifies the record.

All locks acquired by an action are held until the completion of that action, a simplification of standard two-phase locking [9]. 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. As was mentioned above, we make data resilient by storing it on stable storage devices. Furthermore, we do not actually copy information to stable storage until actions commit. Therefore, versions made for a running action and information about locks can be kept in volatile memory. This volatile information 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 standard 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 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. It also reduces 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 will be quite short compared to realistic mean time between failures 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 [14]. We believe that indivisibility is the desired property for most applications, *if* it is required only at the appropriate levels of abstraction. ARGUS provides a mechanism for *user-defined* atomic data types. These types present an external interface that supports indivisibility but that can offer a great deal of concurrency as well. We do not present our mechanism here; user-defined atomic types are discussed in [30].

2.2 Nested Actions

So 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 action. An action may contain any number of subactions, some of which may be performed sequentially, some concurrently. This structure cannot be observed from outside; that is, 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 a subaction commits, aborting its parent action will abort it. Further, object versions are written to stable storage only when top-level actions commit.

Nested actions aid in composing (and decomposing) activities in a modular fashion. This allows a collection of existing actions to be combined into a single, higher level action, and to be run concurrently within that action with no need for additional synchronization. For example, consider a database replicated at multiple nodes. If only a majority of the nodes need to be read or written for the overall action to succeed, this is accomplished by performing the reads or writes as concurrent subactions, and committing the overall action as soon as a majority of the subactions commit, even though some of the other subactions are forced to abort.

Nested actions have been proposed by others [4, 10, 26]; our model is similar to that presented in [23]. 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 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. Since versions become permanent only when top-level actions commit, the two-phase commit protocol is used only for top-level actions. A detailed description of locking and version management in a system supporting nested actions is presented in [23].

In addition to nesting subactions inside other actions, it is sometimes useful to start a new top action inside another action. Such a "nested" top action, unlike a subaction, has no special privileges relative to its parent; for example, it is not able to read an atomic object modified by its parent. Furthermore, the commit of a nested top action is not relative to its parent; its versions are written to stable storage, and its locks are released, just as for normal top actions. Nested top actions are useful for benevolent side effects. For example, in a naming system a name lookup may cause information to be copied from one location to another, to speed up subsequent lookups of that name. Copying the data within a nested top action ensures that the changes remain in effect even if the parent action aborts.

2.3 Remote Procedure Call

Perhaps the single most important application of nested actions is in masking communication failures. Logical nodes (described in Section 3) in ARGUS communicate via messages. We believe that the most desirable form of communcation 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 [16] (see also [29]). 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. In such a case, the communication would fail.[2] The sender can then cope with the failure according to the demands of the particular application. However, coping with the failure is much simpler if it is guaranteed that in this case the remote procedure call had 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 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 caller 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 caller 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: subactions 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, or, in the case of nested top actions, to activities that should not be undone if the parent aborts. 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. Subactions, 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.

Not all effects of an action can be undone by aborting that action, since a change to the external environment, for example, printing a check, cannot be undone by program control alone. But as long as all effects can be undone, the user of our language does not need to write any code to undo or compensate for the effects of aborted actions.

[2] For example, the system would cause the communication to fail if it is unable to contact the remote node. We believe the system, and not the programmer, should take on this kind of responsibility, because the programmer would find it very difficult to define reasonable timeouts.

Before doing something like printing a check, the application program should make sure that printing the check is the right thing to do. One technique for ensuring this is to break an activity into two separate, sequential top-level actions. All changes to the external environment are deferred to the second action, to be executed only if the first action is successful. Such a technique will greatly decrease the likelihood of actions having undesired effects that cannot be undone.

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 of all later actions that read its results) must be defined and executed by the user. Compensation must also be performed for effects of aborted actions that cannot be undone. Note that, in general, there is no way that such compensation could be done automatically by the system, since extrasystem activity is needed (e.g., cancelling already issued checks).

Given our use of a locking scheme to implement atomic objects, it is 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. Rather than having the system prevent, or detect and break, deadlocks, we rely on the user to time out and abort top-level actions. These timeouts generally will be very long, or will be controlled by someone sitting at a terminal. Note that such timeouts are needed even without deadlocks, since there are other reasons why a top action may be too slow (e.g., contention).

A user can retry a top action that aborted because of a timeout or crash, but ARGUS provides no guarantee that progress will be made. ARGUS will be extended if needed (e.g., by raising the priority of a top action each time it is repeated [27] or by using checkpoints [10]).

3. LINGUISTIC CONSTRUCTS

In this section we describe the main features of ARGUS. The most novel features are the constructs for implementing guardians, the logical nodes of the system, and for implementing actions, as described in the previous section. To avoid rethinking issues that arise in sequential languages, we have based ARGUS on an existing sequential language. CLU [17, 20] 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.

3.1 Overview

In ARGUS, a distributed program is composed of a group of *guardians*. A guardian encapsulates and controls access to one or more resources, for example, databases or devices. A guardian makes these resources available to its users by providing a set of operations called *handlers*, which can be called by other guardians to make use of the resources. The guardian executes the handlers, synchronizing them and performing access control as needed.

Internally, a guardian contains data objects and processes. The processes execute handlers (a separate process is spawned for each call) and perform background tasks. Some of the data objects, for example, the actual resources, make up the *state* of the guardian; these objects are shared by the processes. Other objects are local to the individual processes.

A guardian runs at a single node but can survive crashes of this node (with high probability). Thus, the guardians themselves are resilient. A guardian's state consists of *stable* and *volatile* objects. Resiliency is accomplished by writing the stable objects to stable storage when a top action commits; only those objects that were modified by the committing action need be written. The probability of loss of volatile objects is relatively high, so 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, for example, an index into a database for fast access.

After a crash of the guardian's node, the language support system recreates the guardian with the stable objects as they were when last written to stable storage. A process is started in the guardian to recreate the volatile objects. Once the volatile objects have been restored, the guardian can resume background tasks and can respond to new handler calls.

Guardians allow a programmer to decompose a problem into units of tightly coupled processing and data. Within a guardian, processes can share objects directly. However, direct sharing of objects between guardians is not permitted. Instead, guardians must communicate by calling handlers, and the arguments to handlers are passed by value: it is impossible to pass a reference to an object in a handler call. 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 nonlocal objects are more expensive to use. A handler call is performed using a message-based communication mechanism. The language implementation takes care of all details of constructing and sending messages (see [11]).

Guardians are created dynamically. The programmer specifies the node at which a guardian is to be created; in this way individual guardians can be placed at the most advantageous locations within the network. The (name of the) guardian and (the names of) its handlers can be communicated in handler calls. Once (the name of) a guardian or one of its handlers has been received, handler calls can be performed on that guardian. Handler calls are location independent, however, so one guardian can use another without knowing its location. In fact, handler calls will continue to work even if the called guardian has changed its location, allowing for ease of system reconfiguration.

Guardians and handlers are an abstraction of the underlying hardware of a distributed system. A guardian is a logical node of the system, and interguardian communication via handlers is an abstraction of the physical network. 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 the calls 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.

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

Fig. 1. Guardian structure.

```
name = guardian [parameter-decls] is creator-names
                                      handles handler-names
{abbreviations}
{[stable] variable-decls-and-inits}
[recover body end]
[background body end]
{creator-handler-and-local-routine-definitions}
end name
```

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 [17, 20].

The first internal part of a guardian is a list of abbreviations for types and constants. Next 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 heap 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.[4]

Guardian instances are created dynamically by invoking creators of the guardian type. For example, suppose a guardian type named *spooler* has a creator with a header of the form

create = **creator**(*dev*: *printer*) **returns** (*spooler*)

When a process executes the expression

spooler$create(pdev)

a guardian object is created at the same physical node where the process is executing and (the name of) the guardian is returned as the result of the call.[5] Guardians can also be created at other nodes. Given a variable *home* naming some node,

spooler$create(pdev) @ home

creates a guardian at the specified node.

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

[4] In languages that are not object oriented, the concept of reachability would still be needed to accommodate the use of explicit pointers.

[5] As in CLU, the notation *t$op* is used to name the *op* operation of type *t*.

initializing the guardian state and then return the guardian object. (Within the guardian, the expression **self** refers to the guardian object.)

Aside from creating a new guardian instance and executing state-variable initializations, a creator has essentially the same semantics as a handler, as described in the next section. In particular, a creator call is performed within a new subaction of the caller, and the guardian will be destroyed if this subaction or some parent action aborts. The guardian becomes permanent (i.e., survives node crashes) only when the action in which it was created commits to the top level. A guardian cannot be destroyed from outside the guardian (except by aborting the creating action). Once a guardian becomes permanent, only the guardian can destroy itself, using a **destroy** primitive.

The **recover** section runs after a crash. Its job is to recreate a volatile state that is consistent with the stable state. This may be trivial, for example, creating an empty cache, or it may be a lengthy process, for example, creating a database index.

After a crash, the system recreates the guardian and restores its stable objects from stable storage. 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.

After the stable objects have been restored, the system creates a process in the guardian to first execute any initializations attached to declarations of volatile variables of the guardian state and then execute the **recover** section. This process runs as a top-level action. Recovery succeeds if this action commits; otherwise, the guardian crashes, and recovery is retried later.

After the successful completion of a creator, 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; examples are given in Section 4. The **background** section is not run as an action, although generally it creates top-level actions to execute tasks, as explained in Section 3.4.[6]

3.3 Handlers

Handlers (and creators), like procedures in CLU, are based on the termination model of exception handling [19]. 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 in both 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,

```
files_ahead_of = handler(entry_id: int) returns (int)
                                         signals (printed(date))
```

[6] 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 or call handlers without first creating a top-level action.

might be the header of a spooler handler used to determine how many requests are in front of a given queue entry. Calls of this handler either terminate normally, returning an integer result, or exceptionally in condition *printed* with a date 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.

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

Given a variable x naming a guardian object, a handler h of the guardian may be referred to as $x.h$. Handlers are invoked using the same syntax as for procedure invocation, for example,

$x.h$("read", 3, *false*)

However, whereas procedures are always executed locally within the current action, and always have their arguments and results passed by *sharing*,[7] handlers are always executed as new subactions, usually in a different guardian, and always have their arguments and results passed by value.

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

(1) A new subaction of the calling action is created.
(2) A message containing the arguments is constructed. Since part of building this message involves executing user-defined code (see [11]), 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 that 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 a very low probability of the call succeeding if it is repeated immediately.
(5) The system creates a process and a subaction (of the subaction in step (1)) 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.
(6) When the handler terminates, a message containing the results is constructed, the handler action terminates, the handler process is destroyed, and the message is sent. If the message cannot be sent (as in step (2) or (4) above),

[7] Somewhat similar to passing by reference. See [17].

the subaction created in step (1) 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 [19]. For example,

```
count: int := spool.files_ahead_of(ent) % normal return
  except when printed(at: date): ...    % exceptional returns
         when failure(why: string): ...
         end
```

Since a new process is created to perform an incoming handler call, guardians have the ability to execute many requests concurrently. Such an ability helps to avoid having a guardian become a bottleneck. Of course, if the guardian is running on a single-processor node, then only one process will be running at a time. However, a common case is that in executing a handler call another handler call to some other guardian is made. It would be unacceptable if the guardian could do no other work while this call was outstanding.

The scheduling of incoming handler calls is performed by the system. Therefore, the programmer need not be concerned with explicit scheduling, but instead merely provides the handler definitions to be executed in response to the incoming calls. An alternative structure for a guardian would be a single process that multiplexed itself and explicitly scheduled execution of incoming calls. We think our structure is more elegant, and no less efficient since our processes are cheap: creating a new process is only slightly more expensive than calling a procedure.

As was mentioned above, the system does not guarantee message delivery; 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 retry handler calls. Rather, as mentioned earlier, user programs should make progress by retrying top-level actions, which may fail because of node crashes even if all handler calls succeed.

3.4 In-Line Actions

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. It is also possible to have an in-line subaction:

enter action *body* **end**

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

When the body of an in-line action completes, 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 in-line 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.

3.5 Concurrency

The language as defined so far allows concurrency only between top actions originating in different guardians. The following statement form provides more concurrency:

coenter {*coarm*} **end**

where

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

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

The process executing the **coenter**, and the action 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.[8] 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 by the *armtag*.

A simple example making use of **foreach** is in performing a write operation concurrently at all copies of a replicated database:

```
coenter
  action foreach db: db_copy in all_copies(···)
    db.write(···)
  end
```

This statement creates separate processes for the guardian objects yielded by *all_copies*, each process having a local variable *db* bound to a particular guardian. Each process runs in a newly created subaction and makes a handler call.

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 other active coarms of the **coenter** must be terminated. To accomplish this, the system forces all coarms that are not yet completed to abort. A simple example where such early termination is useful is in performing a read operation concurrently at all copies of a replicated database, where a response from any single copy will suffice:

```
coenter
  action foreach db: db_copy in all_copies(···)
    result := db.read(···)
    exit done
  end except when done: ... end
```

[8] An iterator is a limited kind of coroutine that provides results to its caller one at a time [17, 20].

Once a read has completed successfully, the **exit** will commit the read and abort all remaining reads. The aborts take place immediately; in particular, it is not necessary for the handler calls to finish before the subactions can be aborted. (Such aborts can result in *orphan* handler processes that continue to run at the called guardians 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 **background** section of a guardian. In this form the *armtag* can be **process** or **topaction**. The semantics is as above, except that no action is created in the **process** case.

3.6 Program Development and Reconfiguration

ARGUS, like CLU, provides separate compilation of modules with complete type checking at compile time (see [17]). Separate compilation is performed in the context of a program library, which contains information about abstractions (e.g., guardian types).

Before creating a guardian at a node, it is first necessary to load the code of that guardian at that node. Once the code image has been loaded, any number of guardians of that type can be created at that node. It is also possible to load a different code image of the same guardian type at the node, and then create guardians that run that code.

To build a code image of a guardian definition, it is necessary to select implementations for the data, procedural, and iteration abstractions that are used, but not for other guardian abstractions. In other words, each guardian is linked and loaded separately. In fact, each guardian is independent of the implementation of all other guardians, because our method of communicating data values between guardians is implementation independent (see [11]). A guardian is also independent of all abstractions except for those it actually uses. New abstractions can be added to the library, and new implementations can be written for both old and new abstractions, without affecting any running guardian.

Guardians are constrained to communicate with other guardians only via handlers whose types were known when the guardian was compiled. Communication via handlers of unknown type is not sensible; the situation is exactly analogous to calling a procedure of unknown type. Of course, a guardian or handler argument of known type but unknown value can be very useful. We *do* provide this: guardians and handlers can be used as arguments in local procedure calls and in handler calls.

Compile-time type checking does *not* rule out dynamic reconfiguration. By receiving guardians and handlers dynamically in handler calls, a guardian can communicate with new guardians as they are created or become available. For example, the ARGUS system contains a distributed *catalog* that registers guardians and handlers according to their type. The catalog would respond to a request for printer guardians by returning all guardians of type "printer" that previously had been registered.

In many applications it will be necessary to change the implementations of running guardians. We are investigating a replacement strategy that permits new implementations to be provided for running guardians without affecting the users of these guardians [2]. This system also allows for certain kinds of changes in guardian type (e.g., additional handlers).

4. A SIMPLE MAIL SYSTEM

In this section we present a simple mail system, designed somewhat along the lines of Grapevine [1]. This is a pedagogical example: we have chosen inefficient or inadequate implementations for some features, and have omitted many necessary and desirable features of a real mail system. However, we hope it gives 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 locations are 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 there are 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 are used, for example, one at each physical node. All mailers would be registered in the catalog for dynamic lookup. 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.

Two built-in atomic types are used in implementing the mail system: *atomic_array* and *struct*. 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* makes a complete copy of an array. A read lock on the entire array is obtained by *new*, *elements*, and *copy*, and a write lock is obtained by *addh* and *trim*. Structs are immutable (hence atomic) records: new components cannot be stored in a struct object once it has been created. However, the fact that a struct is immutable does not prevent its component objects from being modified if they are mutable.

The mailer guardian is presented in Figure 2. Each mailer is given a registry when created; this registry is the mailer's stable reference to the entire mail system. The mailer also keeps a volatile reference, representing the "best" access

```
mailer = guardian is create
                handles send_mail, read_mail, add_user,
                        add_maildrop, add_registry, add_mailer
reg_list = atomic_array[registry]
msg_list = atomic_array[message]
stable some: registry     % stable reference to some registry
best: registry            % volatile reference to some registry
recover
  best := some            % reassign after a crash
  end
background
  while true do
    enter topaction
      regs: reg_list := best.all_registries( )
      coenter
        action foreach reg: registry in reg_list$elements(regs)
          reg.ping( )    % see if it responds
          best := reg    % make it best
          exit done      % abort all others
        end except when done: end
      end except when failure(*): end
    sleep(···)     % some amount of time
    end
  end
create = creator(reg: registry) returns (mailer)
  some := reg
  best := reg
  return(self)
  end create
send_mail = handler(user: user_id, msg: message) signals (no_such_user)
  drop: maildrop := best.lookup(user)
    resignal no_such_user
  drop.send_mail(user, msg)
  end send_mail
read_mail = handler(user: user_id) returns (msg_list) signals (no_such_user)
  drop: maildrop := best.lookup(user)
    resignal no_such_user
  return(drop.read_mail(user))
  end read_mail
add_user = handler(user: user_id) signals (user_exists)
  drop: maildrop := best.select(user)
    resignal user_exists
  regs: reg_list := best.all_registries( )
  coenter
    action
      drop.add_user(user)
    action foreach reg: registry in reg_list$elements(regs)
      reg.add_user(user, drop)
    end
  end add_user
add_maildrop = handler(home: node)
  drop: maildrop := maildrop$create( ) @ home
  regs: reg_list := best.all_registries( )
  coenter
    action foreach reg: registry in reg_list$elements(regs)
      reg.add_maildrop(drop)
    end
  end add_maildrop
add_registry = handler(home: node)
  new: registry := best.new_registry(home)
  regs: reg_list := best.all_registries( )
  coenter
    action foreach reg: registry in reg_list$elements(regs)
      reg.add_registry(new)
    end
  end add_registry
add_mailer = handler(home: node) returns (mailer)
  m: mailer := mailer$create(best) @ home
  return(m)
  end add_mailer
end mailer
```

Fig. 2. Mailer guardian.

path into the system. The **background** code periodically polls all registries; the first to respond is used as the new best registry.

A mailer performs a request to send or read mail by first using the best registry to look up the maildrop for the specified user and then forwarding the request to that maildrop. A mailer adds a new user by first calling the registry *select* handler to make sure the user is not already present and to choose a maildrop; then, concurrently, the new user/maildrop pair is added to each registry, and the new user is added to the chosen maildrop. A maildrop (or registry) is added by creating the maildrop (or registry) and then concurrently adding it to all registries. A new mailer is created with the current best registry for its stable reference.

Figure 3 shows the registry guardian. The state of a registry consists of an atomic array of registries together with a *steering list* associating an array of users with each maildrop. When a registry is created, it is given the current steering list and an array of all other registries, to which array it adds itself. The *lookup* handler uses linear search to find the given user's maildrop. The *select* handler uses linear search to check if a user already exists, and then chooses some existing maildrop. The *add_user* handler uses linear search to find the specified maildrop and then appends the user to the associated user list. The *add_user*, *add_maildrop*, and *add_registry* handlers perform no error checking because correctness is guaranteed by the mailer guardian.

The maildrop guardian is given in Figure 4. The state of a maildrop consists of an atomic array of mailboxes; a mailbox is represented by a struct containing a user_id and an atomic array of messages. A maildrop is created with no mailboxes. The *add_user* handler is used to add a mailbox. Note that this handler does not check to see if the user already exists since the mailer will have already performed this check. 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; again, the mailer guarantees this.

Now that we have all of the pieces of the mail system, we can show how the initial configuration of the mail system is created:

```
reg: registry := registry$create(reg_list$new( ), steer_list$new( )) @ home1
m: mailer := mailer$create(reg) @ home2
```

where *reg_list* and *steer_list* are defined as in the registry. The resulting mailer can then be placed in the catalog and used to add maildrops and users, as well as more registries and mailers.

Finally, we show a simple use of the mail system, namely, sending a message to a group of users, with the constraint that the message be delivered either to all of the users or to none of them:

```
enter action
  coenter
    action foreach user: user_id in user_group("net")
      m.send_mail(user, msg)
    end except when no_such_user, failure(*):     % ignore failure string
                    abort leave
                 end
  end
```

```
registry = guardian is create
                   handles lookup, select, all_registries, ping,
                           add_user, add_maildrop, new_registry,
                           add_registry
reg_list   = atomic_array[registry]
steer_list = atomic_array[steering]
steering   = struct[users: user_list,     % users with mailboxes
                    drop: maildrop]       % at this maildrop
user_list  = atomic_array[user_id]
stable regs: reg_list          % all registries
stable steers: steer_list      % all users and maildrops
create = creator(rlist: reg_list, slist: steer_list) returns (registry)
  reg_list$addh(rlist, self)     % add self to list
  regs := rlist
  steers := slist
  return(self)
  end create
lookup = handler(user: user_id) returns (maildrop) signals (no_such_user)
  for steer: steering in steer_list$elements(steers) do
    for usr: user_id in user_list$elements(steer.users) do
      if usr = user then return(steer.drop) end
      end
    end
  signal no_such_user
  end lookup
select = handler(user: user_id) returns (maildrop) signals (user_exists)
  for steer: steering in steer_list$elements(steers) do
    for usr: user_id in user_list$elements(steer.users) do
      if usr = user then signal user_exists end
      end
    end
  return(···)      % choose, for example, maildrop with least users
  end select
all_registries = handler( ) returns (reg_list)
  return(regs)
  end all_registries
ping = handler( )
  end ping
add_user = handler(user: user_id, drop: maildrop)
  for steer: steering in steer_list$elements(steers) do
    if steer.drop = drop
      then user_list$addh(steer.users, user)      % append user
           return
      end
    end
  end add_user
add_maildrop = handler(drop: maildrop)
  steer: steering := steering${users: user_list$new( ),
                               drop: drop}
  steer_list$addh(steers, steer)
  end add_maildrop
new_registry = handler(home: node) returns (registry)
  reg: registry := registry$create(regs, steers) @ home
  return(reg)
  end new_registry
add_registry = handler(reg: registry)
  reg_list$addh(regs, reg)
  end add_registry
end registry
```

Fig. 3. Registry guardian.

```
maildrop = guardian is create
                    handles send_mail, read_mail, add_user
box_list = atomic_array[mailbox]
mailbox = struct[mail: msg_list,     % messages for
                 user: user_id]      % this user
msg_list = atomic_array[message]
stable boxes: box_list := box_list$new( )
create = creator( ) returns (maildrop)
  return(self)
  end create
send_mail = handler(user: user_id, msg: message)
  for box: mailbox in box_list$elements(boxes) do
    if box.user = user
      then msg_list$addh(box.mail, msg)     % append message
           return
      end
    end
  end send_mail
read_mail = handler(user: user_id) returns (msg_list)
  for box: mailbox in box_list$elements(boxes) do
    if box.user = user
      then mail: msg_list := msg_list$copy(box.mail)
           msg_list$trim(box.mail, 1, 0)     % delete messages
           return(mail)
      end
    end
  end read_mail
add_user = handler(user: user_id)
  box: mailbox := mailbox${mail: msg_list$new( ),
                           user: user}
  box_list$addh(boxes, box)
  end add_user
end maildrop
```

Fig. 4. Maildrop guardian.

The message is sent to all users simultaneously. A nonexistent user or a failure to send a message transfers control outside the **coenter**, forcing termination of all active coarms; the outer action is then aborted, guaranteeing that none of the messages is actually delivered.

4.1 Remarks

One obvious problem with the mailers as implemented is that, if the best registry for a mailer goes down, the mailer effectively goes down as well, since every task the mailer performs (including choosing a new *best* registry) requires communication with that registry. A better implementation might be for each mailer to have stable and volatile references to multiple registries, and for mailer handlers to try several registries (sequentially) before giving up.

Close examination of the mail system reveals places where the particular choice of data representation leads to 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 will lock out all *lookup* operations.

In a traditional mail system this lack of concurrency might be tolerable, but there are other, similar systems where it would not be acceptable. What is needed are data types that allow more concurrency than do atomic arrays. For example, an 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 do not present implementations here.

The concurrency that *is* built in to the mail system can lead to a number of deadlock situations. For example, in the registry guardian, any two concurrent *add__user* or *add__registry* requests will almost always deadlock, and two *add__maildrop* requests can deadlock by modifying registries in conflicting orders. Some of these deadlocks would disappear if data representations allowing more concurrency were used. For example, the use of a highly concurrent associative memory for the steering list would allow all *add__maildrop* requests to run concurrently, as well as all *add__user* requests for distinct users. Other deadlocks can be eliminated simply by reducing concurrency. To avoid deadlocks between *add__registry* requests, all *new__registry* calls could be made to a distinguished registry, and *new__registry* could obtain a write lock on the registry list before creating the new 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 to 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 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 one of the registries, and using a background process in that registry to (eventually) inform all other registries.

5. SUMMARY AND CONCLUSIONS

ARGUS has two main concepts: guardians and actions. Guardians maintain local control over their local data. The data inside a guardian are 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 starts 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.

ARGUS is quite different from other languages that address concurrent or distributed programs (e.g., [3, 7, 12, 24]). Those languages tend to provide modules that bear 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 data consistency or resiliency. Indeed, the languages completely ignore the problem of hardware failures. 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.

Although a great many details have been omitted, we hope enough of the language has been described to show how ARGUS meets the requirements stated in the introduction. Consistency, service, distribution, concurrency, and extensibility are all well supported in ARGUS. However, there are two areas that are not well supported. One is protection. Guardians could check for proper authorization before performing requests, for example, by requiring principal IDs as arguments to handler calls. But, there is no way within the language to express constraints as to where and when guardians may be created. For example, 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. These kinds of protection issues are under investigation.

Another area that may need work is support for scheduling. Within a guardian a separate process is automatically created for each handler call. This structure provides no direct support for scheduling incoming calls. If one wanted to give certain incoming calls priority over others, this could be done explicitly (by means of a shared monitorlike [13] object). If one wanted certain incoming calls to take priority over calls currently being executed, this could be done (very awkwardly) by programming handlers to relinquish control periodically. However, if one wanted to make priorities global to an entire node, rather than just within a single guardian, there would be no way to accomplish this in ARGUS. We are not convinced that priorities are required frequently enough to justify any additional mechanism. We prefer to adopt a "wait-and-see" attitude, although we are investigating priority mechanisms.

Supporting atomic activities as part of the semantics of a programming language imposes considerable implementation difficulties. We have completed a preliminary, centralized implementation of the language, ignoring difficult problems such as lock propagation and orphan detection. We are working on a real, distributed implementation. At this point it is unclear how efficient such an implementation can be.

The approach to resiliency taken in ARGUS represents an engineering compromise given the current state of hardware. If ultrareliable hardware does become practical, it may no longer be necessary to compensate for hardware failures in software. This would simplify the structure of guardians since stable objects and the recover section would no longer be needed. Furthermore, the implementation of ARGUS would become more efficient.

However, regardless of advances in hardware, we believe atomic actions are necessary and are a 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. For some applications, actions simply may be a convenient tool, not a strictly necessary one. We believe that actions can be implemented efficiently enough that they will be used in applications even when they are not strictly necessary. We expect to get a much more realistic idea of the strengths and weaknesses of the language once the distributed implementation is complete and we can run applications.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the contributions made by members of the ARGUS design group, especially Maurice Herlihy, Paul Johnson, and Bill Weihl. The paper was improved by the comments of the referees and many others.

REFERENCES

1. BIRRELL, A.D., LEVIN, R., NEEDHAM, R.M., AND SCHROEDER, M.D. Grapevine: An exercise in distributed computing. *Commun. ACM 25*, 4 (Apr. 1982), 260-274.
2. BLOOM, T. Dynamic Module Replacement in a Distributed Programming Environment. Ph.D. dissertation, Laboratory for Computer Science, Massachusetts Inst. of Technology, Cambridge, Mass., to appear.
3. BRINCH HANSEN, P. Distributed processes: A concurrent programming concept. *Commun. ACM 21*, 11 (Nov. 1978), 934-941.
4. DAVIES, C.T. Data processing spheres of control. *IBM Syst. J. 17*, 2 (1978), 179-198.
5. DAVIES, C.T., JR. Recovery semantics for a DB/DC system. In Proceedings, ACM 73: Annual Conference, Aug. 1973, pp. 136-141.
6. ESWARAN, K.P., GRAY, J.N., LORIE, R.A., AND TRAIGER, I.L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624-633.
7. FELDMAN, J.A. High level programming for distributed computing. *Commun. ACM 22*, 6 (June 1979), 353-368.
8. GRAY, J.N. Notes on data base operating systems. In *Lecture Notes in Computer Science*, vol. 60: *Operating Systems, An Advanced Course*, R. Bayer, R.M. Graham, G. Seegmüller (Eds.). Springer-Verlag, New York, 1978, pp. 393-481.
9. 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. In *Modeling in Data Base Management Systems*, G.M. Nijssen (Ed.). Elsevier North-Holland, New York, 1976.
10. GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery manager of the System R database manager. *Comput. Surv.* (ACM) *13*, 2 (June 1981), 223-242.

11. HERLIHY, M., AND LISKOV, B. A value transmission method for abstract data types. *ACM Trans. Program. Lang. Syst. 4*, 4 (Oct. 1982), 527–551.
12. HOARE, C.A.R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666–677.
13. HOARE, C.A.R. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct. 1974), 549–557.
14. LAMPORT, L. Towards a theory of correctness for multi-user data base systems. Rep. CA-7610-0712, Massachusetts Computer Associates, Wakefield, Mass., Oct. 1976.
15. LAMPSON, B., AND STURGIS, H. Crash recovery in a distributed data storage system. Xerox PARC, Palo Alto, Calif., Apr. 1979.
16. LISKOV, B. On linguistic support for distributed programs. In Proceedings, IEEE Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh, Pa., July 1981, pp. 53–60.
17. LISKOV, B., ATKINSON, R., BLOOM, T., MOSS, E., SCHAFFERT, J.C., SCHEIFLER, R., AND SNYDER, A. *Lecture Notes in Computer Science*, vol. 114: *CLU Reference Manual.* Springer-Verlag, New York, 1981.
18. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. In Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages, Albuquerque, N.M., Jan. 25–27, 1982, pp. 7–19.
19. LISKOV, B., AND SNYDER, A. Exception handling in CLU. *IEEE Trans. Softw. Eng. SE-5*, 6 (Nov. 1979), 546–558.
20. LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug. 1977), 564–576.
21. LISKOV, B., AND ZILLES, S.N. Programming with abstract data types. In Proceedings, ACM SIGPLAN Conference on Very High Level Languages. *SIGPLAN Notices* (ACM) *9*, 4 (Apr. 1974), 50–59.
22. LOMET, D. Process structuring, synchronization, and recovery using atomic actions. In Proceedings of an ACM Conference on Language Design for Reliable Software. *SIGPLAN Notices* (ACM) *12*, 2 (Mar. 1977).
23. MOSS, J.E.B. Nested Transactions: An Approach to Reliable Distributed Computing. Ph.D. dissertation and Tech. Rep. MIT/LCS/TR-260, Laboratory for Computer Science, Massachusetts Inst. of Technology, Cambridge, Mass., 1981.
24. PRELIMINARY ADA REFERENCE MANUAL. *SIGPLAN Notices* (ACM) *14*, 6 (June 1979), pt. A.
25. RANDELL, B. System structure for software fault tolerance. *IEEE Trans. Softw. Eng. SE-1*, 2 (June 1975), 220–232.
26. REED, D.P. Naming and Synchronization in a Decentralized Computer System. Ph.D. dissertation and Tech. Rep. MIT/LCS/TR-205, Laboratory for Computer Science, Massachusetts Inst. of Technology, Cambridge, Mass., 1978.
27. ROSENKRANTZ, D.J., STEARNS, R.E., AND LEWIS, P.M., II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst. 3*, 2 (June 1978), 178–198.
28. SHRIVASTAVA, S.K., AND BANATRE, J.P. Reliable resource allocation between unreliable processes. *IEEE Trans. Softw. Eng. SE-4*, 3 (May 1978), 230–240.
29. SPECTOR, A.Z. Performing remote operations efficiently on a local computer network. *Commun. ACM 25*, 4 (Apr. 1982), 246–260.
30. WEIHL, W., AND LISKOV, B. Specification and implementation of resilient, atomic data types. Computation Structures Group Memo 223, Laboratory for Computer Science, Massachusetts Inst. of Technology, Cambridge, Mass., Dec. 1982.

Received November 1981; revised September 1982; accepted November 1982