Atomic Data Abstractions in a Distributed Collaborative Editing System[1] (Extended Abstract)

Irene Greif
Robert Seliger[2]
William Weihl

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts

Abstract

This paper describes our experience implementing CES, a distributed Collaborative Editing System written in Argus, a language that includes facilities for managing long-lived distributed data. Argus provides *atomic actions*, which simplify the handling of concurrency and failures, and mechanisms for implementing *atomic data types*, which ensure serializability and recoverability of actions that use them. This paper focuses on the support for atomicity in Argus, especially the support for building new atomic types. Overall the mechanisms in Argus made it relatively easy to build CES; however, we encountered interesting problems in several areas. For example, much of the processing of an atomic action in Argus is handled automatically by the run-time system; several examples are presented that illustrate areas where more explicit control in the implementations of atomic types would be useful.

1. Introduction

As distributed configurations of high-powered workstations connected by networks become prevalent, tools for writing distributed programs take on increasing importance. This paper describes our experience implementing CES, a distributed Collaborative Editing System. CES was written in Argus, a language that was designed to support the construction of reliable distributed programs. Argus provides *atomic actions*, which simplify the handling of concurrency and failures. Atomicity is ensured by *atomic data types*; Argus provides some built-in atomic types, along with mechanisms to permit users to build new atomic types. Our focus in this paper is on the support for atomicity in Argus, especially the support for building new atomic types. Our goal is to evaluate the expressive power provided by Argus and to develop a better understanding of the requirements of distributed applications.

Our analysis of Argus is based on three examples taken from CES, a collaborative editing system developed by the second author [14]. We originally chose Argus as an implementation language for CES because it provides a fast prototyping environment that would allow us to test our co-authorship system on real users, and then refine the design based on their input. Argus did prove to be an excellent development and debugging environment: the system manages network communications and installation of processes at remote sites, and provides a powerful set of distributed debugging tools. In addition, atomic actions proved to be very convenient for

[1]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 Number N00014-83-K-0125, and in part by the National Science Foundation under grant DCR-8510014.

[2]On leave from Hewlett-Packard

controlling concurrency and handling failures such as site crashes.

Although many aspects of Argus proved extremely useful, we encountered interesting problems in several areas. A question raised by all three examples is how much of the processing of an atomic action should be done automatically by the language and run-time system, and how much should be coded explicitly by the programmer. For example, no user code runs in Argus when an action commits or aborts. The examples below illustrate that the lack of explicit control over parts of action processing can lead to program structures that are awkward, indirect, or inefficient.

Another problem involves the way in which the Argus system propagates information about the commits and aborts of multi-site atomic actions from site to site. This problem is illustrated by the second example below. In this case it appears that the current semantics of Argus make it impossible to meet plausible application requirements.

A third area in which we encountered problems is performance. In our initial experiments with the editor, response time was slow enough for the editor to be unusable in realistic testing. When we ran the editor on machines with more physical memory, however, we found that response time was sluggish, but the editor was usable. Thus, it appears that inadequate physical memory was one of the main causes of the initial performance problems. The sluggish response time when running with adequate physical memory can be attributed to a number of causes. For example, the current Argus implementation is an untuned prototype that is built on top of a kernel that in many ways is not well suited to its needs. Also, the structure of the editor itself may be a source of some overhead. For example, the editor checkpoints changes to stable storage every few keystrokes. Most editors in use today save buffers to a file much less frequently than every few keystrokes, so perhaps it is not surprising that CES appears somewhat sluggish in comparison. The current implementation of stable storage may intensify this problem, since stable storage is accessed across a network, more than doubling the access time [11]. As of this writing, we are in the process of conducting further experiments to gain a better understanding of the precise causes of the performance problems.

In the next section we provide an overview of Argus, focusing on the aspects that are most relevant to our examples. In Section 3 we describe the functionality of the collaborative editing system and the organization of its implementation. Next, in Section 4 we present three examples -- display management, version stacks, and document management -- that illustrate the problems mentioned above. We also discuss possible solutions. Finally, in Section 5 we summarize our conclusions and make some suggestions for further work.

2. An Overview of Argus

Argus is a programming language designed to support the construction of reliable distributed systems. It is targeted primarily at applications in which the manipulation and preservation of long-lived online data are of principal importance. CES is one such application; others include banking systems, airline reservation systems, database systems, and various components of operating systems. A major issue in such systems is preserving the consistency of online data in the presence of concurrency and hardware failures. Argus provides mechanisms, discussed below, to help the programmer cope with concurrency and failures. It is based on CLU [10]; as with CLU, abstraction, especially data abstraction, plays a key role in the methodology Argus is designed to support.

2.1. Guardians

A distributed system in Argus is composed of a collection of *guardians*. A guardian is a resource that is resilient to crashes and that can be accessed using remote procedure calls. The definition of a guardian includes several components:

- variables, which refer to objects that represent the state of the guardian;

- creators, which are used to create new instances of the guardian;
- operations (called *handlers*), which can be invoked by other guardians to examine or modify the state of the guardian, and which are the only means by which one guardian can manipulate another guardian's state;
- a background process, which can be used to perform periodic tasks; and
- recovery code, which is used to restore the guardian to a consistent state after a crash.

The background process in a guardian is started automatically when the guardian is created and when it recovers from a crash.

A guardian's variables can be designated as either *stable* or *volatile*. Objects reachable from the stable variables are kept on stable storage [7], which is extremely likely to survive site failures. When a guardian's site crashes, its stable variables are restored from stable storage. The volatile variables are useful for redundant information, like an index for a database, that is easy to reconstruct after a crash. After a crash and before the background process or handlers are allowed to run, the recovery code in a guardian reinitializes the volatile variables.

A guardian can be viewed as a kind of virtual node. Each guardian has a separate address space of objects (similar to a CLU or Lisp heap). These objects are completely local to their guardian; other guardians can access them only through their guardian's handlers, which are call using *remote procedure call*. Arguments to a remote procedure call are passed by value, so that there is no direct sharing between the address spaces of distinct guardians. Data can be shared among sites, however, at the level of a guardian. A guardian is itself an object; the value of a guardian is simply its name, so passing a guardian as an argument of a remote procedure call makes it possible for the receiver of the call to access the guardian.

2.2. Atomic Actions

Concurrency and failures can be handled in Argus by making the activities that use and manipulate data *atomic*. Simple syntax is used to indicate that a sequence of statements should be executed atomically. Atomic activities, or *actions* as they are called in Argus, were first identified in work on databases. Actions are characterized by two properties: serializability and recoverability. *Serializability* means that the concurrent execution of a group of activities is equivalent to some serial execution of the same activities. *Recoverability* means that each activity appears to be all-or-nothing: either it executes successfully to completion (in which case we say that it *commits*), or it has no effect on data shared with other activities (in which case we say that it aborts).

Atomicity simplifies the problem of maintaining consistency by decreasing the number of cases that need to be considered to understand the behavior of a program. Since aborted activities have no effect, and every concurrent execution is equivalent to some serial execution, consistency is ensured as long as every possible serial execution of committed activities maintains consistency. Thus, atomicity simplifies the visible failure modes of a system, and makes it possible to ignore concurrency when checking for consistency.

Atomicity ensures that if an activity cannot complete successfully, it can abort and have no effect. To ensure that the effects of committed actions survive site crashes, the state of a guardian is kept on stable storage. The Argus system ensures that modifications made by an action to any objects accessible through the stable variables of a guardian are saved on stable storage before the action commits.

Argus also supports nested actions (or *subactions*), which can be used to obtain concurrency within a single action and to isolate the effects of failures. For example, a remote procedure call in Argus is executed as a subaction of the calling action; as a result, the call appears to occur exactly zero or one times. If a failure occurs during the call and causes the call to be

aborted, the calling action need not be aborted. Instead, it can try an alternative if one exists.

The main difference between a top-level action (one with no parent) and a subaction is that when a subaction commits, its effects need not be saved on stable storage. Instead, if a crash causes the effects of a committed subaction to be lost, the effects of the subaction can be undone everywhere by aborting the action's parent or some other ancestor. A two-phase commit protocol is used when a top-level action commits to ensure that the effects of the action and all its descendants are saved on stable storage before the action commits; if some of the effects have been lost in a crash, the action is forced to abort.

2.3. Guardians versus Clusters

Argus provides two mechanisms for implementing data abstractions. The cluster, borrowed from CLU, is used to implement a data type each of whose objects belongs to a single guardian. A cluster consists of a description of the new type's representation, along with implementations for each of the operations provided by the type. A cluster's operations are called using *local procedure call*: arguments are passed by sharing (as in CLU and Lisp), the call executes in the same guardian as its caller, and no subaction is created for the call. A cluster's objects cannot be the target of remote procedure calls.

A guardian also implements a kind of data type. It differs from the data types implemented by clusters in that its objects are always remote, and its operations are called using remote procedure call. Since remote access is likely to be significantly more expensive than local access, the designer of a distributed application will have to think carefully about the distribution of data in the system. Thus, the primary difference between a cluster and a guardian -- one provides local objects, and the other remote objects -- arises naturally in the design of a distributed system. Other differences between guardians and clusters, however, can force a designer to use a guardian where a cluster might be more appropriate. For example, a guardian can be active, in the sense that it can have a background process. However, there is no similar way of obtaining a process for a local object. A guardian can also include recovery code for restoring its representation to a consistent state after a crash; no similar capability is available for a cluster. The examples below illustrate the problems caused by these differences between guardians and clusters.

2.4. User-defined Atomic Data Types

Atomicity of activities in Argus is ensured by atomic data types, whose operations provide appropriate synchronization and recovery for actions using objects of the type. Synchronization for the built-in atomic types in Argus is accomplished using strict two-phase locking [2, 8] with read and write locks. The usual semantics applies: read locks can be shared, but write locks conflict with read and write locks. Recovery for the built-in atomic types is ensured by making a copy of an object the first time an action executes an operation that changes the object's state, and then making any changes on the copy. If the action aborts, the copy is discarded; if the action commits, the original version is discarded and replaced with the changed copy.

The built-in types permit relatively little concurrency among actions. Argus also provides mechanisms for implementing new, highly concurrent atomic types [20]. These mechanisms are "implicit," in the sense that no user code is run when actions commit and abort. Instead, synchronization and recovery must be accomplished by including some built-in atomic objects at some level of the representation of a user-defined atomic object.

One important question in the design of a language for distributed programs is how much of the processing of an atomic action should be handled automatically by the system and how much should be handled explicitly by the programmer. The Argus system handles many things automatically: it keeps track of the sites visited by an atomic action and the objects used at each site, it handles the details of the two-phase commit used to ensure that the outcome of the action is recorded consistently at all sites,

and it manages the locks and versions for built-in atomic objects. It is clear that the programmer does not want to handle most of these issues; for example, the details of the two-phase commit protocol and keeping track of the sites visited by an action are easily handled by the system, and there is no apparent reason for letting the programmer handle them more directly. However, the examples below show that more explicit control over the processing of commits and aborts at each object (i.e., lock and version management) would be useful.

Weihl [18, 19] has already observed some limitations of Argus's implicit handling of commits and aborts, and has proposed an alternative structure in which the programmer can provide explicit commit and abort operations as part of the implementation of a data type. Each operation on the type must inform the run-time system when an action uses an object; when the action later commits or aborts, the system invokes the commit or abort operation as appropriate. The problems with Argus's implicit approach illustrated by the first two examples below are nicely solved by Weihl's approach. The examples also illustrate other areas where more explicit control might be useful.

3. Distributed Editor: Functionality and Design

CES is a document editor that supports the collaboration by a group of authors on a shared document. A CES document consists of a structural component, viewed by the author as an outline, and a set of textual components, referred to as the document "nodes". The nodes are arbitrarily sized blocks of text. A readable view of the document is built by ordering the nodes according to the outline in the structural component. The operations in CES extend those of a conventional real-time editor with functions for creating and manipulating structured documents and for modifying the structure of a document independently of the text.

CES is meant to be used in a distributed environment and allows sharing of documents among multiple authors. Each document node is "owned" by an individual author. All authors share access to the document structure, but each is the primary author for his own nodes. An author's nodes reside at his own machine, so that the text of the document is physically distributed across all machines of all co-authors. To improve availability, a copy of a document's structure is kept at all of the sites with access to the document. The copies of this replicated data are kept consistent as users make local changes.

While authors are working on separate nodes of the document, all can be working independently. An author can read any node of a document at any time. If someone else is writing that node, the reader will see a slightly out-of-date version of the node -- CES coordinates the author's activities and tries to minimize the delay in making new versions available for reading. If two or more authors try to write in the same node at the same time, synchronization facilities are invoked to prevent inconsistencies in the text by locking out all but one author.

The nature of this application is such that an author could accidentally keep a document section locked for an arbitrarily long time. For example, he might receive a phone call and stop editing for a while. To protect against unintentional holding of locks, "tickle" locks were designed to be held for as long as some editing activity continues, and to be released if requested by a co-author after an idle period. Rather than abort changes made by the original holder of the lock, small actions are committed during the time that the lock is held, and all of these actions remain visible when the tickle lock is released. The correct scope of such small actions may vary for different situations and different users [3]. The system currently commits user activities after certain "significant" editing commands such as word deletions and carriage returns.

When authors find themselves examining the same node of a document, they may want to coordinate their work more closely, perhaps even shifting into a real-time meeting [12, 13] in which a group of co-authors talk to each other over a voice connection while viewing the document on their individual screens. To support this, we do

not prevent reading of text that is being modified. Instead, screens of all readers are updated at regular intervals as each small action commits.

4. Examples from the Editor

In this section we present three examples of user-defined data types to illustrate the problems we encountered in using Argus to build CES. In the first example, the physical screen is encapsulated in an abstraction whose job is to refresh the screen when an action that wrote on the screen aborts. The techniques available in Argus for detecting that an action has aborted and then taking appropriate action are indirect and awkward to use. A user-defined abort operation would give a simpler and more direct solution. The first example also illustrates a problem arising from the differences between guardians and clusters. A guardian can have a background process, but there is no simple way of associating a background process with a local object.

In the second example, we consider a data type designed to permit a high level of concurrency among users of a document. As in the first example, the implicit handling of commits and aborts leads to awkward program structures that could be avoided by using a more explicit approach. In addition, the way in which Argus propagates information about the commits and aborts of actions from site to site does not provide a sufficiently strong semantics to permit us to meet certain application requirements.

In the third example, we consider a large data structure that is kept on stable storage. Because stable storage is relatively slow, it is important to minimize the amount of data written to stable storage as part of the commit of each action. Argus permits objects to be designated as either stable or volatile, however, only at the level of a guardian; if a local object is designated as stable, then its entire representation is kept on stable storage. In addition, recovery code to reconstruct or reinitialize the volatile parts of an object can only be provided for a guardian. This example suggests that these facilities would also be useful in a cluster.

4.1. Displaying Text

The display buffer and the physical screen for a CES user are encapsulated in a single abstract object, the *display*. The job of the *display* object is to keep the screen consistent with the contents of the buffer. As a user edits a document, the objects representing the document are modified. At the same time, text is written to that user's display buffer, and a side-effect is made visible to the user: text appears on the screen. This immediate feedback is required to keep the user apprised of the system's response to keyboard input. Each change to the document (and the corresponding change to the display) is made as part of an atomic action. A user sees his changes on his screen as he types characters; however, if several users are editing the same document, one user's changes do not become visible to other users until the atomic action in which they are made commits. If this atomic action instead aborts, it is necessary to restore the first user's screen to its state at the start of the action. To accomplish this, we need to be able to detect that an action has aborted and to take appropriate action to refresh the screen. Since no user code runs when an action commits or aborts, indirect methods must be used to detect aborts.

The display object is implemented as an Argus guardian. The state of the display guardian includes two counters. The first, the *commit_count*, is used to keep track of the number of actions that have used the display and committed. The second, the *action_count*, is used to keep track of the number of actions that have used the display, regardless of whether they committed or aborted. The state of the guardian also contains a lock that is acquired by every action that uses the display, and released when the action commits or aborts. If the *action_count* is greater than the *commit_count*, then either some action is currently using the display or some action used the display and aborted. We can tell whether an action is currently using the display by testing the lock in the guardian's state. Thus, we can detect that an action has used the display and aborted.

The background process in the display guardian

is dedicated to the task of checking for aborted actions and, if necessary, refreshing the screen. When an abort is detected and the screen has been refreshed, the *action_count* and *commit_count* are reset to indicate that all aborts have been processed.

Unfortunately, the only way for the background process to detect aborts in Argus is for it to busy-wait, checking periodically whether an abort has occurred. To avoid the overhead of busy-waiting, we added a new type, called a *trigger_queue*, to Argus. A process can call an operation to *wait* on a trigger_queue, causing the process to be blocked until another process calls an operation to *wake up* the waiting process. (This data type could not be implemented in the language itself, since Argus contains no primitives that permit one process to wake up another process. We will return to this issue in Section 4.4.)

The trigger_queue is used in the following manner to avoid busy-waiting. The background process in the display guardian begins by waiting on a trigger_queue. When an action invokes a handler to use the guardian, it acquires the lock in the guardian's state, and then wakes up the background process. When the background process is scheduled to run (which might be immediately after it is awakened and might be at some later time depending on how the system happens to schedule processes), it also attempts to acquire this lock. If the handler action has not yet committed or aborted, the background process will be blocked, waiting for the lock, until the action (not just the handler) completes. If the action has completed, the background process checks whether the action had aborted, and if so it refreshes the screen. It then waits again on the trigger_queue. With this program structure, the background process only wakes up when a handler action starts to use the guardian, and only checks whether an action has aborted after the action has actually completed. Thus, the likelihood that the background process will do unnecessary work is significantly less than it would be if we used busy-waiting.

Using a background process to detect aborts, however, has other problems besides the overhead of busy-waiting. First, in Argus a background process can only be defined as part of the background code of a guardian. This makes it difficult to encapsulate the entire implementation of a type whose objects need a background process in a single module, unless that module is a guardian. In other words, it is difficult to build a cluster-based type, whose objects are local to a guardian, and associate a background process with each of the type's objects.

In the prototype of CES that was built, the display abstraction is a guardian, but this choice would have to be reconsidered to permit more flexible use of windows on the display. We might desire to manage each window on the display separately. Each window should be a local object in a single screen manager, so we would define a cluster-based *window* data type to handle window management. If, however, we need to use a background process to detect the aborts of actions that use windows, each window object must be known to some background process in the guardian. This means that any code that creates a window object must also record the object in some global data structure in the guardian so that the appropriate background process can find it. Furthermore, the window abstraction must provide operations that permit a background process to detect aborted actions and refresh windows on the screen. The modularity of the system would be improved if these details of using windows were hidden from their users.

Second, there are timing problems with using a background process to detect aborts. If the background process does not wake up immediately after an action that used the display aborts, another action might attempt to use the display before the background process detects the abort and refreshes the screen. This means that each action that uses the display must check before updating the screen whether an earlier action had aborted, and then refresh the screen if necessary. We cannot eliminate the background process, however, since if no new action attempts to use the display for a long time, we need the background process to ensure that the screen is

refreshed quickly. Thus, responsibility for detecting aborts and refreshing the screen cannot be allocated to a single piece of code or a single process.

The ability to associate a background process with a local object within a guardian, rather than just with the guardian itself, would avoid the modularity problems discussed above. However, the timing problems would not be solved. In addition, using a background process for each window object could be a source of performance problems.

If the programmer could define explicit commit and abort operations as part of the implementation of each type, the abort operation could refresh the screen as needed. With this approach, there would be no need for a background process or for busy-waiting; instead, the abort operation would run only when needed. The modularity problems with multiple windows would be avoided, since there is no need for a background process, and hence no need for coordination between the *window* type and the guardian in which it is used. The timing problems mentioned above would also be avoided if the commit and abort operations explicitly release locks, rather than having the system release locks automatically as is currently the case in Argus.

4.2. Version Stacks

CES maintains a stack of versions of each document node as it is modified by the various co-authors. The version stack is used to log changes by different authors and to allow an author to back up to a previous version. Each version stack provides operations to push a new version onto the stack, to pop a version off the stack, to read the top of the stack, and to reset the stack (flushing the current contents and pushing a single new entry). A checkpoint can be taken by pushing a new version onto the stack and then modifying that version; operations since the last checkpoint can be undone by popping the top version off the stack. Version stacks are atomic, so modifications to a version stack do not become permanent until the action that made them commits. Thus, until an action commits, changes made by the action can be undone simply by aborting the action. The backup capability provided by a version stack is useful for undoing a sequence of operations that is longer than a single atomic action.

One of the goals of CES is to permit each author to read the entire document, even while other authors are editing parts of the document. Each author would like to see recent changes made by other authors. However, if one author is in the middle of some changes to a node, other authors should not be permitted to read what might be an inconsistent state of the node. In such a situation authors read a version that is not being modified by another author but is as close to the top as possible. An extra operation on version stacks, *fasttop*, is provided for this purpose. The specification of the fasttop operation is nondeterministic: the version returned is not necessarily the top one, but is guaranteed to be no older than one returned in a previous call unless there has been an intervening pop or reset operation. This specification permits more concurrency among actions than would an ordinary "top" operation. In particular, one action can execute fasttop while another action executes push or pop as long as the version returned by fasttop is not the pushed or popped version.

The implementation of the version stack follows the paradigm for highly concurrent atomic types in Argus, such as the *semiqueue* type, defined in [20]. The representation of a version stack consists of a non-atomic sequence of atomic objects. The non-atomic sequence object in the representation is used to achieve the concurrency permitted by the type's specification; the existence of this non-atomic object in the representation is not visible outside the implementation, so at the abstract level version stacks appear atomic to their users. The objects in the sequence must be atomic objects to ensure that modifications made by aborted actions appear to be undone.

When an action modifies a version stack, it may simply modify an atomic object in the representation of the stack (e.g., when popping a

version off the stack), or it may create a new atomic object and add it to the sequence (e.g., when pushing a new version onto the stack). If the action later aborts, any modified atomic objects are restored to their previous states. Modifications to the non-atomic sequence, however, are not undone. Instead, the atomic objects added to the sequence by the aborted action are placed in a state that allows other actions to detect that the objects' creator aborted, and to act as if they were not present in the sequence at all.

In implementing the version stack, we encountered two problems. The first is once again related to the inability to write explicit commit and abort procedures for new types. The second involves the way Argus propagates information about aborts and commits of actions from site to site.

The first problem is that the representation of a version stack gradually accumulates objects that do not represent useful data. For example, as mentioned above, when an action adds a new atomic object to the representation and then aborts, the atomic object is no longer needed. However, it still uses space in the representation. To prevent the representation of a version stack from growing arbitrarily large with such useless components, it is necessary to find and discard such objects.

This kind of garbage collection of representations is typical of implementations of user-defined atomic types in Argus; we have observed it in many other examples (e.g., see [18, 20]). It can be accomplished by cleaning up the representation as part of some or all of the operations on the object, or by using a background process that performs this task periodically. Using a background process for this purpose has the same problems as for detecting aborts. Cleaning up the representation as part of the operations, however, also has problems. Scanning the representation to find useless components imposes some overhead, so it should not be done too frequently. It should also not be done too infrequently, however, since then the representation will grow and the operations will take longer to run. It can be difficult to decide how frequently, and as part of which operations, this cleanup task should be performed. Using Weihl's alternative approach, in which the programmer provides commit and abort operations that are executed automatically by the system, it is possible to remove data from the representation of an object exactly when it is no longer needed, rather than having to notice at some later time that the data is no longer needed and then discard it.

The second problem reveals itself in some surprising behavior visible on the screen to end-users of CES. Suppose the user is working on one machine, and part of the document library is stored on another machine. The user could make a change to a document node stored on the second machine in one action. Once that action has committed, the user could ask to see that part of the document (using the fasttop operation). If the machine on which the document node is stored does not yet know that the first action committed, the fasttop operation might return an older version of the node. The user knows that the node has been changed and that the modifications have been committed, but until the commit event is known at all machines involved he may see information that is out of date.

The delays that result in this behavior are due to the way in which commits and aborts of actions are processed by the Argus system. When an action commits or aborts, the event is recorded locally on the machine where the action is running, but is not necessarily communicated immediately to other machines at which the action (or its subactions) might have run. If the action holds a lock on other machines and another action tries to acquire the lock, the Argus system will send query messages to other machines to find out the outcome of the action holding the lock. If the action that tries to acquire the lock uses an operation that tests the lock but does not wait for it, however, the action will be told that the lock is unavailable. Such tests are common in implementations of user-defined atomic types; for example, the fasttop

operation scans the representation of the version stack looking for a component atomic object that is not locked.

There are two ways in which the semantics of Argus could be changed to solve this problem. One is to change the operation that tests whether a lock is held so that rather than always returning immediately, it waits until it receives a message in response to its query. This response could indicate that the action that holds the lock is still active, or that it has committed, or that it has aborted. If the action has committed or aborted, the lock can be released in the appropriate manner. If the action is still active, then the action that is testing the lock should be informed that the lock is still held. The problem with this approach is that the delay until a response is received could be long. Furthermore, it is difficult to know how long to wait before deciding that the other machine must be down or that the network must be broken. In addition, if we decide to stop waiting, it is not clear what answer to give the action that is testing the lock.

The second solution is to require that information about commits and aborts be propagated among machines more quickly. We could require that if there is a chain of events leading from the commit or abort of an action to a test for a lock held by that action, then the test must indicate that the lock is no longer held. (By "chain of events" we mean events connected by the "happens before" relation of [6], and including events on a single machine and messages over the network.) The difficulty with this approach is that it is not clear whether it can be implemented efficiently enough. It appears to require that each machine keep track of all the actions known by it to have committed or aborted, and that this information be propagated on all messages.

4.3. Document Library

As mentioned earlier, a CES document consists of a structural component, viewed by the author as an outline, and a set of textual components, referred to as the document "nodes". The nodes are arbitrarily sized blocks of text. The CES document library is a collection of documents whose storage is distributed among guardians on each author's site. In each guardian, the contents of the library are kept on stable storage to protect the data against crashes. If an atomic action modifies the library, the modified objects must be copied to stable storage by the time that the action commits. If a guardian containing part of the document library crashes, the copies of objects on stable storage are used to restore the objects to their most recent committed state.

Stable storage is expensive, and relatively slow compared to virtual memory. Thus, it is important to minimize the total amount of data kept on stable storage, and to copy as little data as possible when a given action commits. Argus allows objects to be partitioned into pieces that are copied to stable storage independently, so that only those pieces that are actually modified by an action need to be copied when the action commits. Also, recall that the state of a guardian can be partitioned into *stable* and *volatile* variables, so that information that can be reconstructed after a crash need not be kept on stable storage. (Of course, there is a trade-off here, since reinitializing the volatile variables of a guardian may cost more than keeping them on stable storage.)

The representation of a CES document node contains several fields: a unique identifier; a version stack, which contains old versions of the node for backing up over a scope larger than a single atomic action; and a tickle lock, which consists of the name of the user holding the lock and the time at which it was last "tickled." Some of this information does not need to be recorded on stable storage. For example, tickle locks are intended to be released whenever a guardian crashes, so there is no need to record the state of a tickle lock on stable storage.

The mechanisms in Argus can be used to avoid writing the entire representation of a node to stable storage, but it is awkward to do so. There are two possible approaches. The first is to use the partitioning of a guardian's variables into stable and volatile subsets. Since each document node is identified by a unique identifier, the tickle locks for nodes could be maintained in a separate

table that maps node identifiers to tickle locks and is kept in a volatile variable. Whenever a node is used, the table of tickle locks in the guardian's state must be accessed to check and update the node's tickle lock.

The second approach is to use the *mutex* type in Argus. A mutex object is essentially a container for another object. The mutex object itself performs several functions. First, it can be used to ensure mutual exclusion among processes using the contained object. Second, each distinct mutex object is written independently to stable storage. Furthermore, when an action commits, a mutex object is only copied to stable storage if the action had executed the *changed* operation (provided by the mutex type) on the object. Thus, if we enclose the tickle lock in the representation of a document node in a mutex object and never call the *changed* operation, the tickle lock will be copied to stable storage only once when it is created and never after that.

Both of these approaches, however, have problems. The problem with the first approach is that whenever a document node is created, a tickle lock must be created for it in the guardian's table. In addition, whenever the node is used, the table must be accessed to get the tickle lock. The variables holding a guardian's state can be accessed directly by code in the guardian, but are not accessible to code in other modules. Instead, the table must be passed as an argument to any code that creates or uses a document node. As in the previous example, this need to coordinate use of local objects with the rest of the code in the guardian leads to a loss of modularity.

The problem with the second approach arises if a node's tickle lock needs to be reinitialized after a crash. The only way of reinitializing an object after a crash is to do it in the recovery code of a guardian. This means that a record of all objects requiring reinitialization must be kept in part of the guardian's state so the recovery code can find the objects. As with the first approach, the part of the guardian's state recording these objects must be passed as an argument to all code that creates a document node (though not to all code that merely uses a document node). In fact, tickle locks do not need to be reinitialized after a crash, so the second approach would work well for CES. Nevertheless, we can easily imagine situations in which this approach would not work.

The problems illustrated by this example are similar to the problem discussed in the previous section, in which a background process can only be obtained as part of a guardian. Recovery code can be written only for a guardian, and objects can be partitioned into stable and volatile sets only at the top level of a guardian's state. This means that it can be difficult to encapsulate all details of an object's implementation inside a single module, unless that module is a guardian. This example suggests that explicit control over recovery would be useful in clusters as well as in guardians.

4.4. Summary

All three examples illustrate problems with the support in Argus for building user-defined atomic data types. The first and second examples illustrate problems that can be solved by providing explicit commit and abort operations as part of the implementation of a data abstraction. The first and third examples also illustrate modularity problems caused by the differences between guardians and clusters.

We can imagine several possible solutions to the problems with crash recovery illustrated by the third example. As mentioned earlier, it seems worth exploring alternative approaches that provide more direct control over how an object is stored on stable storage. Approaches that obviate the need for such fine control are also worth investigating; for example, it may be possible to design a hardware stable storage device with access times comparable to virtual memory. If stable storage were cheap and fast enough, one would not need to be concerned with optimizing its use. It may also be possible to dispense with stable storage altogether by replicating objects on several sites (though such an approach may require complicated recovery algorithms). It is not clear which of these

approaches will lead to the simplest and clearest programs.

The problems with propagation of commit and abort information illustrated by the second example could be difficult to solve. As noted above, a naive approach to implementing a stronger semantics would require inordinate amounts of communication. It is not clear to what extent the communication can be reduced. As an aside, we note that this problem is similar to the orphan detection problem [17, 9]; similar solutions may work here as well.

The lack in Argus of a primitive for one process to awaken another process makes it impossible to program a type such as the *trigger_queue*, and thus forces some applications to use busy-waiting. A signalling primitive was not included in the language primarily because the significant events for synchronizing and scheduling atomic actions are the completion (commit or abort) of other actions. Since no user code runs when actions commit and abort, there is no way for one action to signal another when the first action finishes. Weihl's proposal for explicit commit and abort operations includes a signalling mechanism that provides much finer control over scheduling of actions.

5. Conclusions

The main features of CES were suggested by related work on co-authorship [1, 16, 4] and on systems that support collaboration in other applications such as calendar management [5], real-time conferencing [13] and software development [15]. Most of the details of the design, including the basic structure of documents and the user interface requirements, were set out before we decided to use Argus. Thus the CES experience was not preconceived as an Argus-programming exercise and so provides an objective test case for that programming environment. It is the first large program written in Argus.

The question of how much expressive power to provide in a language is always a difficult one. Much of the processing of an atomic action in Argus is handled automatically by the run-time system. The examples above illustrate that more explicit control over some aspects might be useful. More examples need to be studied to decide exactly how much control is needed and what form it should take. Nevertheless, the examples presented here arose in a real application, and thus indicate that serious attention should be paid to the problems they illustrate.

6. Acknowledgements

We thank Barbara Liskov for her many helpful comments on drafts of this paper, Bob Scheifler and Paul Johnson for their help during the construction of CES, and all the members of the Programming Methodology Group at MIT for their feedback on the ideas presented in this paper.

7. References

1. Englebart, D. C. Toward High-Performance Knowledge Workers. Office Automation Conference Digest, AFIPS, April, 1982, pp. 279-290.

2. Eswaran, K. P., *et al.*. "The notions of consistency and predicate locks in a database system". *Comm. ACM 19*, 11 (November 1976), 624-633.

3. Gifford, D. K. and J. E. Donahue. Coordinating Independent Atomic Actions. Proceedings of the IEEE CompCon85, IEEE, February, 1985, pp. 92-94.

4. Greif, I. Computer Support for Cooperative Office Activities. Proceedings of the 1982 Office Automation Conference, AFIPS, San Francisco, California, April, 1982.

5. Greif, I. Teleconferencing and the Computer-Based Office Workstation. Teleconferencing and Interactive Media '82, Madison, Wisconsin, May, 1982.

6. Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System". *Communications of the ACM 21*, 7 (July 1978), 558-565.

7. Lampson, B. *Lecture Notes in Computer Science.* Volume 105: Atomic transactions. In *Distributed Systems: Architecture and*

Implementation, Goos and Hartmanis, Eds., Springer-Verlag, Berlin, 1981, pp. 246-265.

8. Liskov, B. and R. Scheifler. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs". *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1983), 381-404.

9. Liskov, B. *Lecture Notes in Computer Science.* Volume 190: The Argus Language and System. In *Distributed Systems: Methods and Tools for Specification; An Advanced Course*, Goos and Hartmanis, Eds., Springer-Verlag, Berlin, 1985, pp. 343-430.

10. Liskov, B. *et al.* CLU reference manual. In *Lecture Notes in Computer Science*, Goos and Hartmanis, Ed., Springer-Verlag, Berlin, 1981.

11. Oki, B., B. Liskov, and R. Scheifler. Reliable object storage to support atomic actions. Proceedings of the Tenth ACM Symposium on Operating Systems Principles, Rosario Resort, Washington, December, 1985.

12. Sarin, S. and I. Greif. Software for Interactive On-Line Conferences. Proceedings of the Second Conference on Office Information Systems, ACM, Toronto, Canada, June, 1984, pp. 46-58.

13. Sarin, S. and I. Greif. "Computer-Based Real-Time Conferences". *IEEE Computer 18*, 10 (October 1985), 33-45. Special issue on Computer-Based Multimedia Communication.

14. Seliger, R. The Design and Implementation of a Distributed Program for Collaborative Editing. Master Th., Massachusetts Institute of Technology,September 1985.

15. Sluizer, S. and P. Cashman. XCP: An Experimental Tool for Managing Cooperative Activity. Proceedings of the ACM Computer Science Conference, ACM, New Orleans, LA, March, 1985.

16. Trigg, R. H. *A Network-Based Approach to Text Handling.* Ph.D. Th., University of Maryland, November 1983.

17. Walker, E. Orphan detection in the Argus system. Master Th., Massachusetts Institute of Technology,June 1984. Available as MIT/LCS/TR-326.

18. Weihl, W. E. *Specification and Implementation of Atomic Data Types.* Ph.D. Th., Massachusetts Institute of Technology, March 1984. Available as Technical Report MIT/LCS/TR-314.

19. Weihl, W. Linguistic Support for Atomic Data Types. Proceedings of the Workshop on Persistence and Data Types, Scotland, August, 1985.

20. Weihl, W. and B. Liskov. "Implementation of Resilient, Atomic Data Types". *ACM Transaction on Programming Languages and Systems 7*, 2 (April 1985).