LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Specfication and Implementation of Resilient, Atomic Data Types

Computation Structures Group Memo 223
December 1982

William Weihl
Barbara Liskov

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 no. N00014-75-C-0661, and in part by the National Science Foundation under grant no. MCS79-23769.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Specification and Implementation of Resilient, Atomic Data Types

William Weihl
Barbara Liskov

M.I.T. Laboratory for Computer Science
Cambridge, Massachusetts

ABSTRACT

A major issue in many applications is how to preserve the consistency of data in the presence of concurrency and hardware failures. We suggest addressing this problem by implementing applications in terms of abstract data types with two properties: Their objects are atomic (they provide serializability and recoverability for activities using them) and resilient (they survive hardware failures with acceptably high probability). We define what it means for abstract data types to be atomic and resilient. We also discuss issues that arise in implementing such types, and describe a particular linguistic mechanism provided in the Argus programming language.

1. Introduction

There are many applications in which the manipulation and preservation of long-lived, on-line data is of primary importance. Examples of such applications are banking systems, airline reservation systems, office automation systems, database systems, and various components of operating systems. A major issue in such systems is preserving the consistency of on-line data in the presence of concurrency and hardware failures. This paper is concerned with how to define and implement data objects that help provide needed consistency.

To support consistency it is useful to make the activities that use and manipulate the data *atomic*. Atomic activities are referred to as *actions* or transactions; they were first identified in work on databases [5, 6, 8]. An atomic action is distinguished by two properties, indivisibility and recoverability. *Indivisibility* means that the execution of one action never appears to overlap (or contain) the execution of any other action. One way of achieving indivisibility is to run actions serially. However, greater concurrency is desirable, provided the concurrent actions do not interfere with one another. Non-interference can be guaranteed if the effect of running the actions concurrently is the same as if they had been executed serially in some order. If this condition is true, the actions are said to be *serializable* [8, 23].

Recoverability means that the overall effect of an action is all-or-nothing: either all changes made to the data by the action happen, or none of these changes happen. An action that completes all its changes successfully *commits*. Otherwise it *aborts* and whatever changes it made are discarded. Recoverability allows the user on whose behalf an action runs to decide to discard the effects of that action. In addition, it allows an action to work properly even if there is a hardware failure during its execution and allows the system to abort an action if necessary (e.g., to resolve a deadlock).

Atomicity is actually achieved via the shared data, which must be implemented in such a way that using actions appear to be indivisible and recoverable. Data objects that support atomicity will be referred to as *atomic objects*.

Atomicity alone is not sufficient to provide consistency, because it only concerns running actions. In addition, the data must be *resilient*: the probability of loss of data due to a hardware failure such as a node or media crash must be acceptably small. Resilient data are needed to ensure (with high probability) that effects of committed actions are not lost in crashes that occur later.

Up to now, consistency has been provided only in a few database systems (e.g., see [11]) and file systems [27, 24]. In these systems, implementors of applications are limited to using just the types of atomic objects provided by the system, e.g., data base relations or files. These objects are not always the most convenient (or efficient) for implementing applications. We advocate a different approach: implementors should be free to implement their applications in terms of whatever kinds of data objects they find convenient. However, the data objects chosen must support consistency, i.e., they must be atomic and resilient.

The purpose of this paper is to discuss resilient atomic objects. We begin in Section 2 by defining what it means for an object to be atomic. In Section 3 we introduce a model of computation to serve as a basis for discussing the implementation of atomic objects; this model is taken from the Argus programming language that we are developing [21]. Section 4 is concerned with implementing atomic objects under the assumption that hardware failures never occur. Then in Section 5 we admit the possibility of failures and discuss implementing resilience. Finally we conclude with a discussion of open problems.

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

2. Atomic Types

An atomic data type, like a regular abstract data type [18], provides a set of objects and a set of operations. An atomic type is an abstraction, and hence is described by a *specification*; it may be implemented by a *program*. As with regular abstract types, the operations provided by an atomic type are the only way to access or manipulate the objects of the type. Unlike regular types, however, an atomic type provides serializability and recoverability for actions that use objects of the type. For example, relations in most relational databases provide operations to add and delete tuples, and to test for the existence of tuples; these operations are synchronized (for example, using two-phase locking [8]) and recovered (for example, using logs [11]) to ensure the atomicity of actions using the relations.

A specification of an atomic type describes the behavior of objects of the type as observed (via calls on the operations of the type) by the users of those objects. An important question about the specification of an atomic data type is whether the use of the word "atomic" is justified: Are the type's objects defined to behave in a way that ensures the atomicity of actions using the objects? This question has received intense study for a few types (like files and relations) [23]. Experience with implementing these types has shown that the problem is difficult and subtle [8, 11]. If programmers are to implement new, user-defined atomic types, it is especially important to understand in general what behavior is acceptable for an atomic type. In this section we discuss how to specify atomic types, and what it means for a type to be atomic.

An execution of a sequential system is often viewed as a sequence of operations. The legal executions are described by the specifications of the types that provide the operations; these specifications often consist of preconditions and postconditions for each operation (e.g., see [13]). An execution of a concurrent system is more complicated: each process executes a sequence of operations, but the operations executed by different processes may interleave or overlap. To specify the behavior of a type in a concurrent system, it is often convenient to use the events corresponding to the invocations and returns of operations as the steps of execution sequences. In systems of atomic actions, the events corresponding to the commits and aborts of actions must be considered as well. If the type is to be atomic, the execution sequences must be sufficiently constrained so that the actions in any global execution sequence (one involving actions invoking operations at many objects, possibly of different types) are serializable and recoverable.

We may view a specification of an atomic type T as describing a set S(T) of execution sequences. Similarly, an implementation I of an atomic type, perhaps described by a program in some programming language, may be viewed as describing a set S(I) of execution sequences. We define correctness as follows: An implementation I of a type T is *correct* if S(I) is a subset of S(T). This notion of correctness is analogous to partial correctness for sequential programs: an implementation is constrained to do only what is permitted by the specification, but it may actually do less. In this sense the specification may be viewed as describing permissible concurrency. In fact, it may be impractical or undesirable to build an implementation that allows all the executions permitted by its specification. For example, neither the built-in atomic types in Argus nor the implementation presented later in this paper provides as much concurrency as is permitted by its specification.

Our notion of correctness does not permit a specification to require that certain things must happen; such requirements are called liveness or service properties. It seems useful to specify these properties separately from safety properties such as atomicity. We give an example of a service property in section 4, but do not discuss this general class of properties in detail in this paper.

Note that atomicity of actions is a global property, while atomicity for a type must be a local property: it deals only with the events (invocations and returns of operations and commits and aborts of actions) involving the particular type. Such locality is essential if atomic types are to be specified and implemented independently. There are many possible local atomicity properties that result in global atomicity. In this paper we will describe a particular local atomicity property that is used in the programming language Argus.[1] This property is based on several assumptions about the structure of implementations of types; we discuss these assumptions in section 2.1. Then in section 2.2 we describe the local atomicity property in detail, thus defining what it means for a type to be atomic.

2.1 Assumptions

Implementations of atomic types can be characterized by the time at which atomicity of actions is enforced. *Optimistic* implementations let all operations proceed (except perhaps for some local mutual exclusion) in the hope that atomicity will not be violated; thus the term "optimistic" [15]. If atomicity is violated, actions must be aborted. This must be done by the time they try to commit, and may be done earlier if the violation is detected earlier. In contrast, *pessimistic* implementations let operations proceed only if doing so cannot violate atomicity; otherwise, operations must be delayed. It is not clear which of these approaches is more efficient. In this paper we will assume that a pessimistic approach is used.

Implementations of atomic types can be further characterized by the manner in which serializability of actions is ensured. *Static*, or timestamp-based, implementations assume that a global ordering of actions is given *a priori* by an assignment of a unique timestamp to each action. The implementations then ensure that actions are serializable in the order determined by their timestamps. A example of a static system is described in [25]. *Dynamic* implementations allow the serialization order to be determined by what the actions do. For example, if one action reads some information written by another, then the former action must be serialized after the latter. Each of these approaches has advantages [22, 25]; hybrid approaches may be especially effective [1]. In this paper we will assume that a dynamic approach is used.

We make two additional assumptions about what it means for an action to commit. First, once an action has committed, we assume that it cannot be aborted. This rules out implementations in which aborts can "cascade" to already committed actions (e.g., see [29]). Second, once an action has committed, we assume that its effects are visible to actions that are still running. Thus implementations do not have to retain synchronization information for committed actions.

Finally, we assume that an object cannot control when actions invoke operations on the object. In addition, actions that are not waiting for an operation invocation to return are free to commit at any time. Thus, objects must ensure serializability by delaying the returns of operations invoked by actions, and not by refusing to commit an action when it completes.

---

1. Detailed justification of this and other local properties, including ones based on different or weaker assumptions, is given in [28].

2.2 Atomicity

In writing specifications for atomic types, we have found it helpful to pin down the behavior of the operations, initially assuming no concurrency and no failures, and to deal with concurrency and failures later. In other words, we imagine that the objects will exist in an environment in which all actions are executed sequentially, and in which actions never abort. This approach is particularly useful in reasoning about an action that uses atomic objects. The atomicity of actions means that they are "interference-free," so we can reason about (the partial correctness of) an individual action without considering the other actions that might be sharing objects with it [3]. This reasoning process is essentially the same as for sequential programs; the only information required about objects is how they behave in a sequential environment. As an example, an informal sequential specification of some operations on atomic arrays in Argus is described in Figure 1. These arrays are similar to CLU arrays [20], and bear some resemblance to files in that they can grow and shrink.

Given the sequential specification of the operations of a type, we need to define the execution sequences that are permitted by atomicity. To achieve atomicity, we impose two requirements on these execution sequences. To support recoverability, we require that actions can observe the effects of other actions only if those actions committed.[1] This requirement implies that the results returned by operations executed by one action can reflect changes made by operations executed by other actions only if those actions committed. For example, in an atomic array *a*, if one action performs a *store*(a, 3, 7), a second action can receive the answer "7" from a call of *fetch*(a, 3) only if the first action committed.

This requirement supports recoverability since it ensures that effects of aborted actions cannot be observed by other actions. It also supports serializability, since it prevents concurrent actions from observing one another's changes. However, more is needed for serializability: operations executed by one action cannot invalidate the results of operations executed by a concurrent action. For example, suppose an action A executes the *size* operation on an atomic array object, receiving *n* as the result. Now suppose another action B is permitted to execute *addh*. The *addh* operation will increase the size of the array to *n* + 1, invalidating the results of the *size* operation executed by A. Since A observed the state of the array before B executed *addh*, A must precede B in any sequential execution of the actions (since sequential executions must be consistent with the sequential specifications of the objects). Now suppose that B commits. By assumption, A cannot be prevented from seeing the effects of B. If A observes any effect of B, it will have to follow B in any sequential execution. Since A cannot both precede and follow B in a sequential execution, serializability would be violated.

To state our requirements more concretely, let us begin by considering a simple situation involving two concurrent actions each executing a single operation on a shared atomic object O.

1. Note that this requirement could be weakened to allow an action to observe the effects of another action that had not yet committed, but then an abort of the latter action would force an abort of the former. This would violate our assumption that actions should be free to commit unless they are waiting for a pending invocation, since the former action can commit only if the latter action also commits.

Fig. 1. Partial specification of atomic arrays.

Here T is an arbitrary type.

new( ) **returns** (atomic_array[T])
    effect returns a new, empty atomic array with low bound 0, high bound -1 and size 0.

low (a: atomic_array[T]) **returns** (int)
    effect returns the current low bound of *a*.

high (a: atomic_array[T]) **returns** (int)
    effect returns the current high bound of *a*.

size (a: atomic_array[T]) **returns** (int)
    effect returns the current size of *a*.

addh (a: atomic_array[T], x: T)
    effect extends *a* by appending *x* on the high end; the current high bound and size of *a* increase by 1.

addl (a: atomic_array[T], x: T)
    effect extends *a* by appending *x* on the low end; the current low bound of *a* decreases by 1, and the size of *a* increases by 1.

remh (a: atomic_array[T]) **returns** (T) **signals** (bounds)
    effect if *a* is empty signals bounds. Otherwise removes and returns the element at the high end of *a*; the size and high bound of *a* decrease by 1.

reml (a: atomic_array [T]) **returns** (T) **signals** (bounds)
    effect if *a* is empty signals bounds. Otherwise removes and returns the element at the low end of *a*; the low bound of *a* increases by 1, while the size decreases by 1.

store (a: atomic_array [T], i: int, x: T) **signals** (bounds)
    effect if *i* is outside bounds of *a* signals bounds; otherwise stores *x* in the *ith* location of *a*.

fetch (a: atomic_array [T], i: int) **returns** (T) **signals** (bounds)
    effect if *i* is outside bounds of *a* signals bounds; otherwise returns the object stored in the *ith* location of *a*.

(The actions may be executing operations on other shared objects also: we are defining a local atomicity property, so we focus on the operations involving a single object O.) A fairly simple condition that guarantees serializability is the following. Suppose O is an object of type T. O has a current state determined by the operations performed by previously committed actions. Suppose O1 and O2 are two executions of operations on O in its current state. (O1 and O2 might be executions of the same operation or different operations.) If O1 has been executed by an action A and A has not yet committed or aborted, O2 can be performed by a concurrent action B only if O1 and O2 *commute*: given the current state of O, the effect (as described by the sequential specification of T) of performing O1 on O followed by O2 is the same as performing O2 on O followed by O1. By "effect" we mean both the results returned and any modifications to the state of O.

The intuitive explanation of why the above condition works is as follows. Suppose O1 and O2 are performed by concurrent actions A1 and A2 at O. If O1 and O2 commute, then the order in

which A1 and A2 can be serialized does not matter at O. If A1 is serialized before A2 then the local effect at O is as if O1 were performed before O2, while if A2 is serialized before A1, the local effect is as if O2 were performed before O1. But these two effects are the same since O1 and O2 commute.

Notice that the common method of dividing operations into readers and writers and using read/write locking works because it allows operations to be executed by concurrent actions only when the operations commute. Our condition permits more concurrency than readers/writers because the meaning of the individual operations and the arguments of the calls can be considered. For example, calls of the atomic array operation *addh* always commute with calls of *addl*, yet both these operations are writers. As another example, *store*(O, i, e1) and *store*(O, j, e2) commute if $i \neq j$.

Note that we require that O1 and O2 commute only when they are executed starting in the current state. For example, consider a bank account object, with operations to deposit a sum of money, to withdraw a sum of money (with the possible result that it signals *insufficient funds* if the current balance is less than the sum requested), and to examine the current balance. Two withdraw operations, say for amounts *m* and *n*, do not commute when the current balance is the maximum of *m* and *n*: either operation when executed in this state will succeed in withdrawing the requested sum, but the other operation must signal *insufficient funds* if executed in the resulting state. They do commute whenever the current balance is at least the sum of *m* and *n*. Thus if one action has executed a withdraw operation, our condition allows a second action to execute another withdraw operation while the first action is still active as long as there are sufficient funds to satisfy both withdrawal requests.

Our condition is similar to the commutativity condition identified in [2]. The condition in [2], however, appears to require that O1 and O2 commute in all possible states if they are to be executed by concurrent actions. This condition is more restrictive than ours, and does not permit two actions to execute withdraw operations concurrently. The greater generality of our condition may be important for achieving reasonable performance.

Our condition must be extended to cover two additional cases. First, there may be more than two concurrent actions at a time. Suppose A1,...,An are concurrent actions, each performing a single operation execution O1,...,On, respectively, on O. (As before, the concurrent actions may be sharing other objects as well.) Since A1,...,An are permitted to be concurrent at O, there is no local control over the order in which they may appear to occur. Therefore, all possible orders must have the same effect at O. This is true provided that all permutations of O1,...,On have the same effect when executed in the current state, where effect includes both results obtained and modifications to O.

The second extension acknowledges that actions can perform sequences of operation executions. For example, suppose action A executed *addh* followed by *remh* on an array. This sequence of operations has no net effect on the array. It is then permissible to allow a concurrent action B to execute *size* on the same array, provided the answer returned is the size of the array before A executed *addh* or after it executed *remh*. To extend the definition, consider concurrent actions A1,...,An each performing a sequence S1,...,Sn, respectively, of operation executions. This is permissible if all sequences Si1,Si2,...,Sin, obtained by concatenating the sequences S1,...,Sn in some order, produce the same effect.

The set of execution sequences described by the sequential specification of the operations of a type is thus all sequences that satisfy the condition described in the preceding paragraphs. This condition may be summarized in an inductive fashion as follows, indicating when it is legal to extend an execution sequence by allowing a pending invocation to return: Assume that the currently active actions A1,...,An have already executed sequences S1,...,Sn, respectively, of operation executions, and that some action A has a pending invocation. (Note that A is active since it has a pending invocation.) This invocation may be executed and allowed to return if, starting in the current state (defined by the operations executed by the actions that have already committed), all sequences Si1,Si2,...,Sin, obtained by concatenating the sequences S1,...,Sn in some order (where A's sequence has been extended by appending the new operation execution), produce the same effect.

Note that in requiring certain sequences of operations to have the same effect, we are considering the effect of the operations as described by the specification of the type. Thus we are concerned with the abstract state of O, and not with the concrete state of its storage representation. Therefore, we may allow two operations (or sequences of operations) that do commute in terms of their effect on the abstract state of O to be performed by concurrent actions, even though they do not commute in terms of their effect on the representation of O. This distinction between an abstraction and its implementation is crucial in achieving reasonable performance, and is the basis for the example implementation to be presented in Section 4.

3. Model of Computation

In the remainder of the paper we will be discussing implementation of atomic objects. This discussion must be based on some model of computation. The purpose of this section is to define such a model. Our model is taken from Argus, a new language we are developing that provides atomic objects [21]. Argus is based on a number of decisions about how to implement atomicity and resilience. For example, we use 2-phase locking to order actions, and we keep redundant information in stable storage to achieve resilience. The Argus model satisfies the assumptions stated in Section 2.1; indeed, it is the reason why we made that particular set of assumptions.

Argus is a language and system that supports distributed programs, i.e., programs that run on a distributed hardware base. Each node in this base is an independent computer consisting of one or more processors and some local memory; the nodes may differ in the number and types of processors, the amount of memory, and in the attached I/O devices. The nodes can communicate only by sending messages over the network; we make no assumptions about the network topology. We will assume such a base in this paper because this allows us to address the consistency problem in its most general form.

In Argus, an application is implemented from one or more modules called *guardians*. Each guardian consists of some data objects and some processes to manipulate those objects. The processes within a guardian can share the objects directly, but sharing of objects between guardians is not permitted. Instead a guardian provides access to its objects via a set of operations called *handlers* that can be called from other guardians.[1] Arguments to handler calls are passed by value; it is impossible to pass a reference to an object in a handler call. This rule ensures that all references to an object are within that object's guardian.

Each guardian resides at a single physical node, although a node may support several guardians. Guardians survive crashes

1. A guardian is similar to a Simula class instance [4].

of their node of residence and other hardware failures with high probability, and are therefore resilient. When a guardian's node crashes, all processes within the guardian are lost, but a subset of the guardian's objects, referred to as the guardian's *stable state*, survives. After a crash, the guardian recovers with its stable state intact; it then runs a special recovery process to recover the remainder of its objects. Resilience is accomplished by copying the guardian's stable objects to stable storage [17] periodically, as discussed further in Section 5.

In addition to guardians, Argus also provides atomic actions and atomic data types. Actions are the primary method of carrying out computations in Argus. Actions terminate by either committing or aborting and are indivisible and all-or-nothing provided the only data shared among them is atomic. An action starts at one guardian but can spread to other guardians by means of handler calls. When an action completes it either commits at all guardians or aborts at all guardians. The Argus implementation ensures this atomicity property.

Argus provides a number of built-in atomic data types, for example, atomic arrays and atomic records. We have chosen implementations for the built-in atomic types in Argus that are efficient but provide for less than maximal concurrency. For atomic arrays and the other built-in atomic types operations are classified as readers and writers, and readers and writers exclude one another in the usual way. Thus, for example, the array *addh* and *addl* operations are both writers and therefore exclude each other even though this is not necessary for atomicity.

We use a locking approach to implementing serializability. Atomic arrays and the other built-in types are implemented using strict 2-phase locking [8]. The locks are acquired automatically when a primitive operation (like *addh*) is called by an action, and are held until the calling action terminates (commits or aborts). Recoverability is provided by making a copy the first time an action executes a writer operation. All changes are made to this copy. The copy replaces the original if the action commits; if the action aborts the copy is discarded.

4. User-Defined Atomic Types

The built-in atomic types in Argus are somewhat limited in their provision of concurrency. Users may very well specify new atomic types that permit a great deal of concurrency. If users were constrained to implementing new atomic types only in terms of the built-in atomic types, the desired concurrency could not be achieved.

For example, consider the *semi-queue* data type. Semi-queues are similar to queues except that dequeuing does not happen in strict FIFO order. They have three operations: *create*, which creates a new, empty semi-queue; *enq*, which adds an element to a semi-queue, and *deq*, which removes and returns an arbitrary element *e* that was enqueued previously and has not yet been dequeued.

Semi-queues have very weak concurrency constraints. Two *enq* operations commute with each other, as do an *enq* and a *deq* operation or two *deq* operations as long as they involve different elements. Thus many different actions can *enq* concurrently, or *deq* concurrently. Furthermore one action can *enq* while another *deq's* provided only that the latter not return the newly *enq*'d element.

We impose the following service requirement on semi-queues: *deq* must eventually remove any element *e* that is eligible for dequeuing. We have used the semi-queue in a printer subsystem, in which actions submit files to be printed, and the subsystem prints a file once the action that submitted it has committed. This constraint on the element returned by *deq* is enough for the printer subsystem to guarantee that each file submitted by an action that later commits will eventually be printed.[1]

The semi-queue data type could be implemented using an atomic array as a representation, e.g.,

```
rep = atomic_array[elem]
```

In this case, the implementation of *enq* would simply be to *addh* the new element to the atomic array. Since addh is a writer, an *enq* operation performed on behalf of some action A would exclude *enq* and *deq* operations from being performed on behalf of other actions until A completed. As observed above, the specification of the semi-queue permits much more concurrency than this. Note that the potential loss of concurrency is substantial since actions can last a long time. For example, an action that performed an *enq* may do a lot of other things (to other objects at other guardians) before committing.

To avoid loss of concurrency, it is necessary to provide a way for users to implement new atomic types directly from non-atomic types. Users who implement atomic types will face a number of new problems, however, and these problems will influence the design of a linguistic mechanism to support user-defined atomic types.

To some extent, the issues involved in implementing an atomic type are similar to those that arise in implementing other abstract types. The implementation must define a representation for the atomic objects, and an implementation for each operation of the type in terms of that representation. However, the implementation of an atomic type must solve some problems that do not occur for ordinary types, namely: inter-action synchronization, making visible to other actions the effects of committed actions, and hiding the effects of aborted actions.

A way of thinking about the above set of problems is in terms of events that are of interest to an implementation of an atomic type. Like implementations of regular types, these implementations are concerned with the events corresponding to operation calls and returns; here, as usual, control passes to and from the type's implementation. In addition, however, events corresponding to termination (commit and abort) of actions that had performed operations on an object of the type are also of interest to the type's implementation.

Linguistic mechanisms to support implementation of atomic types can be divided into two categories based on how information about termination events is conveyed to a type's implementation. In the *explicit approach*, an implementation would find out about these events explicitly, e.g., by providing special commit and abort operations that are called by the runtime system when actions commit and abort. Alternatively, in the *implicit approach* an implementation is not informed about action termination, but rather must find out about it after the fact.

It is too early to predict what category of linguistic mechanism will turn out to be more successful. It is even too early to analyze the tradeoffs between the two categories; such analysis must wait until well-engineered examples of the categories are available for study. As always, we can expect the usual goals to be in conflict: ease of use, efficiency, expressive power, and

1. This is not quite true when we consider failures: the action that dequeues a file to print it could abort every time, preventing any progress from being made. As long as failures do not occur sufficiently often to cause this situation, every file will be printed eventually. An interesting open question is how to state service requirements for systems that can fail.

simplicity. For example, an explicit mechanism requires the programmer to provide the additional commit and abort operations, and thus may appear to be less simple and harder to use than an implicit mechanism in which these operations are not needed. However, in an implicit mechanism it may be difficult to obtain needed information about past termination events, and this difficulty could actually cause the mechanism to be harder to use than an explicit mechanism.

4.1 Implementing Atomic Types in Argus

In this section we describe how user-defined atomic types can be implemented in Argus. We view our mechanism as a first attempt to address the issues in this area. Some related work [26] addresses only the problem of achieving proper synchronization of actions, and not recovery or process synchronization issues.

Our mechanism is based on the implicit approach, but it is quite low level, and more complex than we would like. The design was influenced primarily by considerations of efficiency and expressive power, and tradeoffs were made that favored these goals over simplicity, safety and ease of use. Our assumption is that only rarely are new atomic types implemented in terms of non-atomic types and therefore it is reasonable to require considerable sophistication of the programmer attempting such an implementation.

Since we are following the implicit approach, the first question is how do programs find out about commit and abort events? Our answer is: through the use of objects of built-in atomic types. The representation of a user-defined atomic type will therefore be a combination of atomic and non-atomic objects, with the non-atomic objects used to hold information that can be accessed by concurrent actions, and the atomic objects containing information that allows the non-atomic data to be interpreted properly. The built-in atomic objects can be used to ask the following question: Did the action that caused a particular change to the representation commit (so the new information is now available to other actions), or abort (so the change should be forgotten), or is still active (so the information cannot be released yet)? The operations available on built-in atomic objects have been extended to support this type of use, as will be illustrated below.

The use of atomic objects permits operation implementations to discover what happened to previous actions and to synchronize concurrent actions. However, the implementations also need to synchronize concurrent operation executions. Here we are concerned with *process concurrency* (as opposed to action concurrency), i.e., two or more processes are executing operations on the same object at the same time.

We provide process synchronization by means of a new data type called *mutex*. Mutex objects provide mutual exclusion, as implied by their name. A mutex object is essentially a container for another object. This other object can be of any type, and mutex is parameterized by this type. An example is

mutex[array[int]]

where the mutex object contains an array of integers. Mutex objects are created by calling operation

create = **proc** (x: T) **returns** (mutex [T])

which constructs a new mutex object containing *x* as its value. The contained object can be retrieved later via operation

get_value = **proc** (m: mutex [T]) **returns** (T)

This operation delivers the value of the mutex object, namely (a pointer to) the contained T object, which can then be used via T operations. *Get_value* can be called via the syntactic sugar *m.value* where *m* is a mutex object.

The **seize** statement is used to gain possession of a mutex object and (optionally) retrieve the contained object;

seize *expr* [**using** *decl*] **do** *body* **end**

Here *expr* must evaluate to a mutex object. If that object is not now in the possession of a process, this process *gains possession*. Then, if the optional **using** clause is present, the object contained in the mutex is retrieved (via the *get_value* operation) and assigned to the new variable. The process then executes the *body*. Possession is *released* when control leaves the *body*. If some process has possession, this process waits until possession is released.[1] If several processes are waiting, one is selected fairly as the next one to gain possession.

The **seize** statement as explained above is semaphore-like: ignoring the optional **using** clause, it could be translated to

```
P(m.sem)
body
V(m.sem)
```

where *m* is the mutex object obtained by evaluating *expr* and we imagine this object has a semaphore as a component. However, the **seize** statement is more powerful than this because inside its *body* it is possible to release possession temporarily. This is done by executing the **pause** statement:

pause

Execution of this statement releases possession of the mutex object that was obtained in the smallest statically containing **seize** statement. The process then waits for a system determined amount of time, after which it attempts to regain possession; any competition at this point is resolved fairly. Finally, once it gains possession it starts executing in the *body* at the statement following the **pause**.

Clearly the combination of **seize** with **pause** gives a structure that is similar to monitors [14]. However, **pause** is simply a delay; there is no guarantee that when the waiting process regains possession, the condition it is waiting for will be true.[2] The reason why we do not provide an analog of a monitor's condition variables is the following: Often the conditions these processes are waiting for concern commit and abort events. These are not events over which other user processes have any control. Therefore, it would not make sense to expect user processes to signal such information to each other.

4.2 Implementation of Semi-queues

In this section we present an example implementation of the semi-queue data type described earlier. We use this example to illustrate how objects of built-in atomic type can be used to find out about the completion of actions, and how mutex can be used to synchronize user processes.

The implementation appears in Fig. 2. The implementation is simply a CLU cluster [19, 20]. For simplicity we are assuming the elements in the semi-queue are integers. The plan of this implementation is to keep the enqueued integers in a regular (non-atomic) array. This array can be used by concurrent actions, but it is enclosed in a mutex object to ensure proper process

1. A runtime check is made to see if possession is held by this process. In this case, the **seize** statement fails with the exception *failure* ("deadlock").

2. In Mesa [16] there is similarly no guarantee when a waiting process awakens.

Fig. 2. Implementation of the Semiqueue Type

```
semiqueue = cluster is create, enq, deq

qitem = atomic_variant[enqueued: int,
                       dequeued: null]
buffer = array[qitem]
rep = mutex[buffer]

create = proc () returns (cvt)
  return(rep$create(buffer$new()))
  end create

enq = proc (q: cvt, i: int)
  item: qitem := qitem$make_dequeued(nil)    % dequeued if action aborts,
  qitem$change_enqueued(item, i)             % enqueued if commits
  seize q using b: buffer do
    buffer$addh(b, item)                     % add new item to buffer
    end
  rep$changed(q)          % notify system of modification to buffer
                          % (this will be explained later)
  end enq

deq = proc (q: cvt) returns (int)
  cleanup(q)
  seize q using b: buffer do
    while true do
      for item: qitem in buffer$elements(b) do
        % look at all items in the buffer
          tagtest item % see if item can be dequeued by this action
            wtag enqueued (i: int): qitem$change_dequeued(item,nil)
                                    return(i)
            end % tagtest
        end % for
      pause
      end % while
    end % seize
  end deq

cleanup = proc (q: rep)
  enter topaction          % start an independent action
    seize q using b: buffer do
      for item: qitem in buffer$elements(b) do
        % remove only qitems in the dequeued state
        tagtest item
          tag dequeued: buffer$rem(b)
          others: return
          end % tagtest
        end % for
      end % seize
    end % enter -- commit cleanup action here
  end cleanup

end semiqueue
```

synchronization. All modification and reading of the array occurs inside a **seize** statement on this containing mutex object.

To determine the status of each integer in the array, we associate with each integer an atomic object that tells the status of actions that inserted or deleted that item. For this purpose we use the built-in atomic type, *atomic_variant*. Atomic variant objects are similar to variant records. An atomic variant object can be in one of a number of states; each state is identified by a tag and has an associated value. Atomic variant operation *make_t* creates a new variant object in the *t* state; this state is the object's "base" state, and the object will continue to exist in this state even if the creating action aborts. Operation *change_t* changes the state (the tag and value) of the object; this change will be undone if the calling action aborts. There are also operations to decompose atomic variant objects, although these are usually called implicitly via special statements. Atomic variant operations are classified as readers and writers; for example, *change_t* is a writer, while *make_t* is a reader.

In this paper, atomic variant objects will be decomposed using the **tagtest** statement. [1]

tagtest *expr*
  { *tagarm* }
  [**others** : *body*]
  end

where

tagarm :: = *tagtype idn* [(*decl*)] : *body*

tagtype :: = **tag** | **wtag**

The *expr* must evaluate to an atomic variant object. Each *tagarm* lists one of the possible tags; a tag can appear on at most one arm. An arm will be selected if the atomic variant object has the listed tag, and the executing action can obtain the object in the desired mode: read mode for **tag** and write mode for **wtag**. If an arm can be selected, the object is obtained in the desired mode. Then, if the optional declaration is present, the current value of the atomic variant object is assigned to the new variable. Finally, the associated *body* is executed. If no arm can be selected and the optional **others** arm is present, the *body* of the **others** arm is executed; if the **others** arm is not present, control falls through to the next statement. [2]

The semi-queue operations are implemented as follows. The *create* operation simply creates a new empty array and places it inside of a new mutex object. The *enq* operation associates a new atomic variant object with the incoming integer; this variant object will have tag "enqueued" if the calling action commits later, and tag "dequeued" if it aborts. Then *enq* seizes the mutex and adds the new item to the contained array.

The *deq* operation seizes the mutex and then searches the array for an item it can dequeue. If an item is enqueued and the action that called *deq* can obtain it in write mode, that item is selected and returned after changing its status to "dequeued". Otherwise the search is continued. If no suitable item is found, **pause** is executed and later the search is done again.

Proper synchronization of actions using a semi-queue is achieved by using the *qitems* in the buffer. An *enq* operation need not wait for any other action to complete. It simply creates a new *qitem* and adds it to the array. Of course, it may have to wait for another operation to release the mutex object before adding the *qitem* to the array, but this delay should be relatively short. A *deq* operation must wait until some *enq* operation has committed; thus it searches for a *qitem* with tag "enqueued" that it can write.

---

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

2. The **tagtest** statement can be used to discover information about concurrent actions, and thus violate atomicity (although we don't do this in the examples). There is another decomposition statement, **tagwait**, that is safe in the sense that its use cannot violate atomicity.

The *qitems* are also used to achieve proper recovery for actions using a semi-queue. Since the array in the mutex is not atomic, changes to the array made by actions that abort later are not undone. This means that a *deq* operation cannot simply remove a *qitem* from the array, since this change could not be undone if the calling action later aborted. Instead, a *deq* operation changes the state of a *qitem*; the atomicity of *qitems* ensures proper recovery for this modification. If the calling action later commits, the *qitem* will have tag *dequeued* permanently. Such *qitems*, which are also generated by *enq* operations called by actions that later abort, have no effect on the abstract state of the semi-queue. Leaving them in the array wastes storage, so the internal procedure *cleanup*, called by *deq*, removes them from the low end of the array.[1] It seems characteristic of the general approach used here that reps need to be garbage collected in this fashion periodically.

Cleanup cannot run in the calling action because then its view of what the semi-queue contained would not be accurate. For example, if the calling action had previously executed a *deq* operation, that *deq* appears to have really happened to a later operation execution by this action. But of course the *deq* really hasn't happened, because the calling action has not yet committed.

To get a true view of the state of the semi-queue, *cleanup* runs as an independent action. This action has its own view of the semi-queue, and since it hasn't done anything to the semi-queue previously, it cannot obtain false information. The independent action is started by the **enter** statement:

enter topaction *body* **end**

It commits when execution of the *body* is finished.

An independent action like the *cleanup* action commits while its calling action is still active. Later the calling action may abort. Therefore, the independent action must not make any modifications that could reveal intermediate states of the calling action to later actions. The *cleanup* action satisfies this condition because it performs a *benevolent side effect*: a modification to the semi-queue object that cannot be observed by its users.

5. Resilience

Atomicity was discussed in Section 2 as if failures could never occur. However, failures do occur, and, in fact, can never be entirely prevented, which gives rise to two problems: First, how do we extend specifications to describe resilience, and second, how do we implement resilience? This section addresses only the latter problem.

If hardware were highly reliable, i.e., reliable enough that the probability of loss of information were extremely small, then the mechanism described above would be sufficient for implementing resilient atomic objects. It may be that hardware will be this reliable in the future, but not today: node and media failures are likely to lead to the loss of information. Therefore, some method of implementing data resilience in software is needed.

Various methods of achieving resilience might be imagined. For example, every object might have one or more copies existing at different locations in the network, and actions would read and update the multiple copies in accordance with one of the known algorithms, e.g., Gifford's algorithm [9]. If one of the copies became unavailable because of a failure, some method of recovering that copy from one of the existing copies would be needed.

1. A more realistic implementation would call *cleanup* only occasionally.

We are using a different approach based on stable storage [17]. Each object has a backup copy kept on stable storage, which has the property that information entrusted to it is extremely unlikely to be lost. New backup copies must be written to stable storage for all objects modified by committed actions. This copying can happen when the change takes place, or after the change takes place; it must happen before an action can commit.

Objects are kept in volatile memory while they are used by actions, as are locks and the copies made by writers for built-in atomic objects. If a node crashes, the locks and copies will be lost for all objects at that node. In this case, any action that used objects at that node and had not yet committed must be forced to abort. To ensure that the action will abort, a standard two-phase commit protocol [10] 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.

When a physical node crashes, all guardians residing at that node become inaccessible. However, since copies of a guardian's objects reside on stable storage, a guardian is not destroyed by a crash. Instead when a node recovers, its guardians restart; the information in stable storage is used to restore the states of the objects.

Both built-in and user-defined atomic objects must be copied to stable storage when the actions that modified them commit. This requirement raises the question of how the user controls what is written to stable storage. If we were using an explicit approach, the user might provide an operation that the system could call to cause writing to stable storage. However, in our implicit approach we must make do without such an operation. Our solution is to extend the meaning of mutex.

So far, mutex has been used only for synchronization of user processes. Now it will be used for three additional functions: notifying the system when information needs to be written to stable storage, defining what information is written to stable storage, and ensuring that information is written to stable storage in a consistent state.

The system knows when a built-in atomic object has changed: this can happen only if the committing action holds a write lock on the object or created the object. New mutex objects are also written to stable storage when the creating action commits. In addition, we provide mutex operation

changed = **proc** (m: mutex[T])

for notifying the system that an existing mutex object should be written to stable storage. Calling this operation will cause *m* to be written to stable storage by the time the action that executed the *changed* operation commits. Note that *changed* is not really needed; the system could keep track of all mutex objects used by an action (via the *get_value* operation) and write these to stable storage. But we are concerned that writing to stable storage is expensive and therefore should be avoided if possible. The *changed* operation allows the user to avoid copying of mutex objects that need not be copied (e.g., were only read).

Copying a mutex object involves copying the contained object. By choosing the proper granularity of mutex objects the user can control how much is written to stable storage. For example, a large data base can be broken into partitions that are written to stable storage independently by partitioning it among several mutex objects. The *changed* operation can be used to

limit writing to stable storage to just those partitions actually modified by a committing action.

Finally, mutex objects can be used to ensure that information is in a consistent state when it is written to stable storage. The system will gain possession of a mutex object before writing it to stable storage. By making all modifications to these objects inside **seize** statements, the user's code can prevent the system from copying the object when it is in an inconsistent state.

In the semi-queue example in the previous section, the addition of a new *qitem* to the array by an *enq* operation certainly needs to be stably recorded if the calling action commits; otherwise no permanent record of the *enq* operation would exist. Thus the *enq* operation uses the *changed* operation to notify the system of this fact. Then, when the enqueuing action commits, the system writes the array, including the value of the new *qitem*, to stable storage. A *deq* operation modifies an existing *qitem*; this change will be stably recorded since *qitems* are atomic. The effect of a *deq* operation on the array, however, does not need to be stably recorded. A *deq* operation only modifies the array in an invocation of *cleanup*. If these changes are forgotten in a failure that restores an earlier state of the array, the presence of the extra *qitems* in the array will not affect later operations, and *cleanup* will remove them again the next time it is executed. Thus the modification made by *cleanup* need not be recorded stably (though it will be when the next action that executes *enq* commits).

The above discussion of copying to stable storage has ignored two issues that must now be discussed. The first concerns the recoverability of copying mutex objects to stable storage. Clearly, the copying of each individual mutex object must be all-or-nothing. But, can the copying of several mutex objects be all-or-nothing? Our answer is to provide recoverability on a per guardian basis, but not for the system as a whole. Our condition guarantees consistency within each guardian, but not between guardians.

The second issue concerns mutex and built-in atomic objects that refer to one another. Suppose the system is copying a mutex object that contains as a component a mutex or built-in atomic object. Should that contained object be copied to stable storage too? And, if so, in what order are the two objects copied, and, if they are both mutex objects, does the system gain possession of both before copying either?

The method we use for copying data to stable storage has the following properties.

1. It minimizes writing: only those objects actually modified by the committing action are copied.

2. It is incremental: each built-in atomic object and each mutex object is written to stable storage in a separate, atomic step. In copying each such object, the system copies all portions of the object except contained mutex and atomic objects. These are copied separately if they were modified, or if they are new.

3. It is order-independent: the atomic and mutex objects are written to stable storage in an arbitrary order (chosen to increase the efficiency of the system).

Thus, when an *enq* operation commits, the system gains possession of the mutex object, waiting if necessary, and then copies the names (but not the values) of the contained *qitems* to stable storage. In addition, those *qitems* that were modified by the committing action, or that are new (e.g., the newly enqueued *qitem*), are also written to stable storage, but this is done independently of the copying of the array state. In particular, the system does not have possession of the mutex object while copying the *qitems* to stable storage. Furthermore, the order in which these various objects are written to stable storage is undefined; the system might copy the array state first and later a contained modified *qitem*, or vice versa.

Copying to stable storage is incremental for the following reason. The alternative would be to write all modified objects together. To do so the system would have to gain possession of all changed mutex objects before writing any of them. Such a requirement would be likely to delay the system substantially (especially when you consider that the objects are distributed), leading to an unacceptable delay in the execution of the first phase of two-phase commit. In fact it might be impossible for the system ever to obtain all locks. We chose the incremental scheme to avoid such problems.

The incremental scheme has the following impact on programs. The true state of an object usually includes the states of all contained objects, and a predicate expressing a consistency condition on an object state would normally constrain the states of contained objects (this predicate is usually referred to as the *representation invariant* [12]). For example, suppose we had an atomic type *double-queue* that (for some reason) kept two copies of the semi-queue and was represented by

rep = record [first, second: semiqueue]

where the representation invariant required that the states of the two semi-queues be the same. Now suppose the system is handling the commit of some action A that modified both semi-queues contained in the double-queue, and while this is happening a second action B is modifying those semi-queues. Then it is possible that when the *first* semi-queue is written to stable storage it contains B's changes, but when the *second* semi-queue is written to stable storage it does not contain B's changes. Therefore, the information in stable storage appears not to satisfy the representation invariant of the double-queue.

However, the representation invariant of the double-queue really is satisfied, for the following reason. First note that the information in stable storage is only of interest after a crash. So suppose there is a crash. Now there are two possibilities:

1. Before that crash, *B* also committed. In this case the data read back from stable storage is, in fact, consistent, since it reflects *B*'s changes to both the *first* and *second* semi-queues.

2. *B* aborted or had not yet committed before the crash. In either case, *B* aborts. Therefore, the changes made to the *first* semi-queue by *B* will be hidden by the semi-queue implementation: at the abstract level, the two semi-queues *do* have the same state.

The point of the above example is that if the objects being written to stable storage are atomic, then the fact that they are written incrementally causes no problems.

On the other hand, when an atomic type is implemented with a representation consisting of several mutex objects, the programmer must be aware that these objects are written to stable storage incrementally, and care must be taken to ensure that the representation invariant is still preserved and that information is not lost in spite of incremental writing. We have explored several atomic type implementations that use more than one mutex. Often incremental writing is not a problem; for example, this is the case when a database is simply implemented as a number of partitions. If incremental writing must be taken into account, this can be accomplished as in the following example.

Suppose we decide to implement a database by two mutex objects, a log and a table. The table is very large and contains most of the information in the data base. The log contains only a record of recent activity. The true state of the data base is a combination of the information in the log and the table. For example, in responding to a query, the log would be examined first. Only if needed information was not in the log would it be necessary to consult the table. By keeping recent information in the log, only the log need be copied to stable storage when accessing actions commit. Furthermore, locks need be stored only in the log and not in the table.

Since the log must be copied to stable storage frequently, it is important to keep it fairly small. This can be accomplished by running a cleanup action periodically to copy information in the log to the table. The cleanup action has an associated cost of copying the table to stable storage, so we must balance the frequency of the cleanup action against the size of the table. (The table could be partitioned to reduce this cost.) Once information has been moved from the log to the table, it can be deleted from the log. However, care must be taken to ensure that the moved information is recorded in the stable copy of the table before it is deleted from the stable copy of the log. This is accomplished by writing the table, but not the log, to stable storage as part of the commit of the cleanup action. Furthermore, the cleanup action runs inside a seize of the log, which serves to delay writing of the log to stable storage (on behalf of accessing actions) until after the cleanup action has committed. The changes made to the log by the cleanup action will be written to stable storage later, when an accessing action commits.

6. Conclusions

In the preceding sections we discussed atomic types and why user-defined atomic types are needed. We then discussed the issues that arise when users implement atomic types; these issues must be considered in designing a linguistic mechanism to support user-defined atomic types. Finally we presented a linguistic mechanism consisting of the mutex type, the built-in atomic types, and the associated statement forms.

A system design methodology based on the use of atomic actions and atomic types results in systems with useful modularity properties. The partial correctness of an individual action can be verified independently of the other actions in the system and of the implementations of the atomic types. Similarly, an implementation of an atomic type can be verified independently of which actions use objects of the type. This independence is especially useful if a system performs poorly because of internal concurrency limitations. In such a case, it may be possible to identify certain shared objects as bottlenecks, and to substitute more concurrent (albeit more complex) implementations for the types defining those objects. Thus, it may be possible to trade off simplicity for concurrency systematically.

Our approach to implementing user-defined types was designed to provide good expressive power and user control over efficiency. The extent to which we have succeeded in achieving these goals is not yet clear. One difficulty in evaluating expressive power is that there is not yet a set of canonical problems that can be used as a benchmark. Most of the problems we have looked at we have been able to solve efficiently. However, there are problems that are difficult to solve using our mechanism. For example, protocols that process read-only actions more efficiently [1] appear to require extensions to the mechanism. Also, there are complicated locking protocols for search structures [7] that cannot be implemented easily with our mechanism.

In evaluating the ease of use of our mechanism, it is worth discussing atomicity and resilience separately. A problem in implementing atomicity, and one we have not discussed in this paper, is that it is difficult to analyze the starvation and deadlock properties of implementations. To some extent, this difficulty is inherent in the problem domain. However, it is difficult to guarantee fairness with our mechanism, even if mutex is fair. The problem is that the underlying mechanism does not guarantee that when a waiting process awakens its condition for proceeding is satisfied. Therefore, it is possible that each time a process awakens, it must pause again, and so it starves. Fairness can be programmed using our mechanism by keeping a queue of active actions and doing scheduling explicitly, but such a solution is complicated and inefficient.

If we changed our mutex mechanism to link **pause** to a boolean expression that must be true for the process to be awakened, then the system could provide fairness. However, the cost of implementing such a mechanism is substantial, and may not be worthwhile, for two reasons. First starvation probably doesn't happen in practice (see [16]). Second, what is really wanted is some guarantee of progress for the system as a whole, and it is not clear how local guarantees of progress like fairness contribute to global progress, particularly in the presence of failures.

An alternative to the implicit approach is the explicit approach. One interesting fact about Argus is that the system-provided implementation of the built-in atomic types follows the explicit approach. It is worth noting that the system implementations are quite complex, primarily because in Argus we have nested subactions. These are actions that can commit and abort independently of their parent, and that can run concurrently with their siblings. However, the commit of a child subaction is relative to the parent; if the parent aborts the effects of the child will be undone. We are convinced that nested subactions are necessary as discussed in [21]. One reason for choosing an implicit mechanism for Argus is that nested subactions have little impact on it; this is why we did not need to explain nested subactions in this paper. Nested subactions appear to have a major impact on an explicit mechanism. An explicit mechanism, however, may provide a more uniform basis for implementing the specialized protocols (e.g., for read-only actions) mentioned earlier.

In addition to synchronization, mutex is used to provide resilience. It is worth noting that the time may come when resilience is provided by the hardware. It is also true that it may become cost effective to provide resilience in software but below the level of the language. If we could assume resilience at the language level, much of the difficulty in implementing atomic types disappears.

If programmers must cope with resilience, then some notion of copies and copying is necessary. We keep those copies on stable storage; an alternative would be to keep them at other nodes. In either case, the programmer must worry about how the copying happens. Also, efficiency considerations may dictate that the copies be space-efficient.

The main advantage of an explicit approach appears to be to give the user direct control over copying. With such control the user should be able to insure that the copy satisfies the representation invariant, and also contains just the right information. Also, it is probably easier with an explicit approach to use a representation of the data in stable storage that differs from that used in volatile storage. An explicit approach raises a number of difficult questions, however. The code for translating the volatile representation into the stable representation runs during

the first phase of the two-phase commit protocol for an action. Suppose this code makes remote procedure calls to other guardians. Is this allowed? Do those guardians need to be included in the two-phase commit for this action? Concern about such problems was a major factor in our decision to pursue the implicit approach. These questions must be resolved before a complete evaluation of the different approaches can be made.

We believe atomic types are useful for building general systems that depend on consistent on-line data. In this paper we have discussed what makes a type atomic, and the issues that arise in implementing atomic types. More work is needed in investigating these issues and in designing linguistic mechanisms. Our discussion was based on a number of assumptions; different assumptions may lead to changes in the basic definition of atomicity and to different linguistic mechanisms. Our hope in writing this paper is to interest others in this area of research.

7. References

1. Bernstein, P., and Goodman, N., "Concurrency control algorithms for multiversion database systems", *ACM Symposium on Principles of Distributed Computing*, Ottawa, August 1982, 209-215.

2. Bernstein, P., Goodman, N., and Lai, M., "Two part proof schema for database concurrency control", *Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks*, February 1981, 71-84.

3. Best, E., and Randell, B., "A formal model of atomicity in asynchronous systems", *Acta Informatica 16*, 1981, 93-124.

4. Dahl, O.-J., et al., "The Simula 67 common base language", *Publication No. S-22*, Norwegian Computing Center, Oslo, 1970.

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

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

7. Ellis, C., "Concurrent search and insertion in 2-3 trees", *Acta Informatica 14*, 1980, 63-86.

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

9. Gifford, D., "Weighted voting for replicated data", *Proceedings of the Seventh ACM SIGOPS Symposium on Operating Systems Principles*, December 1979, 150-162.

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

11. Gray, J.N., et al. "The recovery manager of the System R database manager", *ACM Computing Surveys 13, 2*, June 1981, 223-242.

12. Guttag, J., Horowitz, E., and Musser, D., "Abstract data types and software validation", *Communications of the ACM 21, 12*, December 1978, 1048-1064.

13. Guttag, J., and Horning, J., "Formal specification as a design tool", *Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages*, January 1980, 251-261.

14. Hoare, C.A.R., "Monitors: an operating system structuring concept", *Communications ACM 17, 10*, October 1974, 549-557.

15. Kung, H.T., and Robinson, J.T., "On optimistic methods for concurrency control", *ACM Transactions on Database Systems 6, 2*, June 1981, 213-226.

16. Lampson, B., and Redell, D., "Experience with processes and monitors in Mesa", *Communications of the ACM 23, 2*, February 1980, 105-117.

17. Lampson, B., "Atomic transactions", Distributed Systems: Architecture and Implementation, *Lecture Notes in Computer Science 105*, Goos and Hartmanis editors, Springer-Verlag, Berlin, 1981, 246-265.

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

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

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

21. Liskov, B., and Scheifler, R., "Guardians and actions: linguistic support for robust, distributed programs", *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, January 1982, 7-19. Revised version to appear in TOPLAS.

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

23. Papadimitriou, C.H., "The serializability of concurrent database updates", *Journal of the ACM 26, 4*, October 1979, 631-653.

24. Paxton, W., "A client-based transaction system to maintain data integrity", *Proceedings of the Seventh ACM SIGOPS Symposium on Operating Systems Principles*, December 1979, 18-23.

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

26. Schwarz, P., and Spector, A., "Synchronizing shared abstract types", *Technical Report CMU-CS-82-128*, CMU Department of Computer Science, September 1982.

27. Sturgis, H., Mitchell, J., and Israel, J., "Issues in the design and use of a distributed file system", *Operating Systems Review 14, 3*, July 1980, 55-69.

28. Weihl, W., PhD thesis, MIT Laboratory for Computer Science, forthcoming.

29. Wood, W.G., "Recovery control of communicating processes in a distributed system", *Technical Report 158*, University of Newcastle upon Tyne, 1980.