DISTRIBUTED PROGRAMMING IN ARGUS

Argus—a programming language and system developed to support the implementation and execution of distributed programs—provides mechanisms that help programmers cope with the special problems that arise in distributed programs, such as network partitions and crashes of remote nodes.

BARBARA LISKOV

Argus—a programming language and system—was developed to support the implementation and execution of distributed programs. Distribution gives rise to some problems that do not exist in a centralized system, or that exist in a less complex form. For example, a centralized system is either running or crashed, but a distributed system may be partly running and partly crashed. The goal of Argus is to provide mechanisms that make it easier for programmers to cope with these problems.

A program in Argus runs on one or more nodes. Each node is a computer with one or more processors and one or more levels of memory; we assume that the nodes are heterogeneous, i.e., contain different kinds of processors. Nodes can communicate with one another only by exchanging messages over the network. We make no assumptions about the network topology; for example, the network might be a local area net, or it might consist of a number of local area nets connected by a long haul net. In such a network, it is usually much faster for a node to access local information than information residing in some other node.

Distributed programs must cope with failures of the underlying hardware. Both the nodes and the network may fail. The only way nodes fail is by crashing; we assume it is impossible for a failed node to continue sending messages on the network. The network may lose messages or delay their delivery or deliver them out of order. It may also partition, so that some nodes are unable to communicate with other nodes for some period of time. In addition, the network may corrupt messages, but we assume the corruption is detectable; this assumption is satisfiable with arbitrarily high probability by including redundant information in messages.

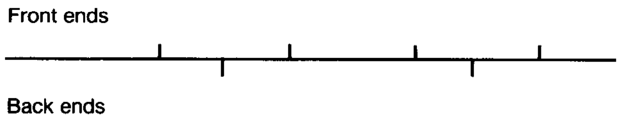
Argus is intended to be used primarily for programs that maintain online data for long periods of time, e.g., file systems, mail systems, and inventory control systems. These programs have a number of requirements. Online information must remain consistent in spite of failures and also in spite of concurrent access. Programs must provide some level of service even when components fail; for example, a program may replicate information at several nodes so that individual failures can be masked. Programmers need to place information and processing at a particular node, both to do replication properly, and to improve performance, since information is cheaper to access if it is nearby. Finally, programs may need to be reconfigured dynamically, by adding and removing components, or by moving a component from one node to another. To minimize the impact of moving components, the method used to access information should be location independent. Argus was designed to satisfy these requirements.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, and monitored by the Office of Naval Research under Contract N00014-83-K-0125, and in part by the National Science Foundation under Grant DCR-8503662.

A DISTRIBUTED BANK

Imagine a large bank with branches in geographically distributed locations. Information about the bank's accounts is stored online. To ensure that information about the accounts of a particular branch can be accessed locally, the online database of accounts is also physically distributed, with information about the accounts of each branch stored at a computer or computers at that branch. Nevertheless, an important goal of the system is to support remote interactions with accounts. For example, a customer whose account is at branch A can make a deposit or withdrawal at branch B and have the amount be credited or debited to his account properly. Also, customers can make withdrawals through cash vending machines located at numerous geographical locations. Furthermore, employees at the bank's main office can access information at all branches for auditing and other management purposes.

The configuration for the banking system is illustrated in Figure 1. Each clerk interacts with a *front-end* program that runs on a minicomputer; a single mini may run the front-end programs for many clerks. Cash vending machines are connected to other programs at the minicomputers. Other users of the system also make use of programs running at the minicomputers. For example, bank personnel can produce monthly statements or run audits by interacting with such programs. The minicomputers are connected via a network to the *back-end* computers where the account information resides. Every mini can communicate with every back end and vice versa.



The front ends are minicomputers that interact with clerks and other users, and control cash vending machines. The back ends are mainframes; each stores information about accounts for some branch and physically resides at that branch. The front ends interact with the back ends to carry out deposits and withdrawals.

FIGURE 1. Configuration of the Banking System

Each back end belongs to a particular branch of the bank: It resides at that branch and maintains a branch database containing information about accounts on site. Information stored for an account includes the account number, the name and address of the owner of the account, the balance, and other information such as a log of all transactions processed against the account.

Example uses of the system are sketched in Figure 2, which shows two procedures that run at the front ends. The first is the *audit* procedure, which allows an administrator to compute the total assets for some subset of the branches; the identities of the branches of interest are passed in as an array. *Audit* sends a request for the current total to each branch of interest and sums the results. This program is object-oriented in the sense that the branch databases are objects that can be asked to perform requests such as deposits and withdrawals; the notation *b.total* means the request for the total should be sent to branch *b*. The work of doing a withdrawal, deposit, or computing a branch's total, is actually performed at the branches themselves.

The *transfer* procedure is used by a clerk to carry out a transfer of funds from one account, referred to as *from*, to another account, the *to* account. This procedure either terminates normally, meaning that the transfer has been successful, or it signals an exception, *insufficient_funds*, if the balance of the *from* account is smaller than the desired amount. The transfer is carried out by withdrawing the desired amount from *from* and depositing it in *to*. First an attempt is made to withdraw the desired amount from the *from* account. This request is directed to *from's* branch. If there are sufficient funds in *from*, then the desired amount is deposited in the *to* account by directing a deposit request to the branch of the *to* account. The function *get_branch* computes an account's branch given its account number. (We will discuss the details of how such a computation might be done later.)

There are a number of problems with the procedures shown, two of which are the following:

1. *Concurrent activities may interfere with one another.* For example, if a transfer runs concurrently with an audit, the audit might record a total that includes the withdrawal but not the deposit.
2. *Various failures are not taken into account.* For example, suppose the back end of the *to* account crashes immediately after the withdrawal from the *from* account has been made. In this case we need to either put the money back into the *from* account or wait to complete the transfer until the *to* account's back end recovers.

ARGUS

Argus was designed to support programs like the banking system. To capture the object-oriented nature of such programs, it provides a special kind of object called a *guardian*, which implements a number of procedures that are run in response to remote requests. To solve the problems of concurrency and failures we have mentioned, Argus allows computations to run as *atomic transactions*, or *actions* for short. We will describe guardians and actions here; however more information can be found in [11, 12, 14]. We will illustrate their uses by showing how a portion of the banking system can be implemented in Argus.

Guardians

An Argus guardian is a special kind of abstract object whose purpose is to encapsulate a resource or resources. It permits its resource to be accessed by means of special procedures, called *handlers*, that can be called

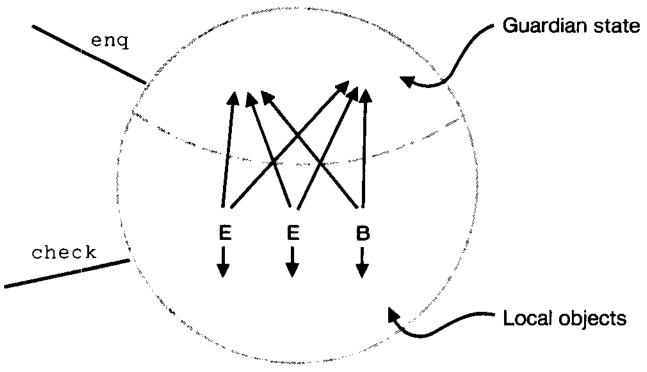
```
audit = proc (branches: array[branch]) returns (int)
      sum: int := 0
      for b: branch in elements(branches) do
            sum := sum + b.total( )
            end
      return (sum)
      end audit

transfer = proc (from, to: account_number, amt: int) signals (insufficient_funds)
      f: branch := get_branch(from)
      t: branch := get_branch(to)
      f.withdraw(from, amt)
            except when insufficient_funds: signal insufficient_funds end
      t.deposit(to, amt)
      end transfer
```

Audit is used to compute the total assets of some of the branches. Transfer is used to transfer money from one account to another. A transfer may work incorrectly if a back end fails while it is running, and it may cause a concurrent audit to produce the wrong total.

FIGURE 2. Two Front-end Procedures

from other guardians. For example, a guardian might encapsulate some or all of the accounts at a branch, and provide handlers to open and close accounts, and to withdraw and deposit money in accounts. As another example, a guardian might control a printing device, and provide a handler called *enq* to allow files to be enqueued for printing and a handler called *check_queue* to check the state of the queue. A printer guardian is illustrated in Figure 3.



The guardian provides a handler to enq a file for printing, and a handler to check the state of the queue. It contains three processes (the triangles): Two are running enq requests, while the third is a background process that is supervising the printing of the current file. The state of the guardian consists of stable objects (the squares), which survive crashes of the guardian's node, and volatile objects (the circles), which do not. A process can access all objects comprising its guardian's state. In addition, processes have local, volatile objects of their own. Only the stable objects survive a crash; after the crash a special recovery process is run to restore the volatile state.

FIGURE 3. The Printer Guardian

A guardian contains within it data objects that store the state of its resource. These objects are not accessible outside the guardian; the only way they can be accessed or modified by another guardian is by calls of their guardian's handlers. Handler calls are performed using a message-based communication mechanism. Arguments are passed by value, which ensures that a guardian's objects cannot be accessed directly by any other guardian. The Argus implementation takes care of all details of constructing and sending messages.

Inside a guardian are one or more processes. Processes can access all the guardian's objects directly. Some processes carry out handler calls; whenever a handler call arrives at a guardian, a process is created to run the call. In addition there may be *background* processes that carry out tasks independently of particular handler calls. For example, the *enq* handler of the printer guardian might merely record information about the request; a background process would carry out the actual printing.

Each guardian resides at a single node of the network, although it can change its node of residence. Several guardians can reside at the same node. A guardian is *resilient* to failures of its node. Some of its objects survive crashes; these are the stable objects, and they are written periodically to stable storage devices. With high probability, stable storage devices avoid loss of information in spite of failures [9]. The other objects in the guardian are volatile. For example, in the printer guardian, information about queued requests would be stored in stable objects so that requests are not lost in a crash. However, detailed information about the exact processing of the current request need not be stable, since the request can be redone after a crash.

A crash destroys all volatile objects of a guardian and also all processes that were running at the time of the crash. After the crash, the Argus system restores the guardian's code and recovers the stable objects from stable storage. Then it creates a special recovery process, which runs code defined by the guardian to initialize the volatile objects. When this process finishes, the guardian is ready to accept new handler calls and to run background processes. Since the volatile state does not survive crashes, it should be used only to record redundant information (e.g., an index into a database) or information that can be discarded in a crash (e.g., current printing information in the printer spooler).

A guardian can create other guardians dynamically and (the names of) guardians and handlers can be sent as arguments of handler calls. The creator specifies the node at which the new guardian is to reside; in this way individual guardians can be placed at the most advantageous locations. Handler calls are location independent, so that one guardian can use another without knowing its location.

A distributed program in Argus is composed of a number of guardians residing at a number of nodes. For example, in the banking system there might be a guardian running at the back-end computer of each branch to carry out requests on the accounts of that branch. In addition, there would be a guardian for each input agent. This guardian would use a background process to listen for input, and then make handler calls to the appropriate branch guardians. The branch guardians would remember all crucial information about accounts in stable objects, so that we can be sure the effect of a withdrawal or deposit is not lost if the branch's computer crashes.

Guardians allow programs to be decomposed into units of tightly coupled data and processing. However, they do not solve the synchronization and failure problems mentioned earlier. These problems are addressed by the second main mechanism in Argus, the atomic action.

Actions

Argus permits a computation such as a transfer or an audit to run as an atomic action [3]. Actions have precisely the properties that are needed to solve the concurrency and failure problems. First, they are *serializable*: the effect of running a group of actions is the same as if they were run sequentially in some order. Second, they are *total*: an action either completes entirely or it is guaranteed to have no visible effect. An action that completes is said to *commit*; otherwise, the action *aborts*.

Serializability solves the concurrency problem. If a transfer action and an audit action are running concurrently, then the effect must be as if they ran sequentially in some order. Either the audit will (effectively) run after the transfer is finished or before it starts; in either case it observes the proper total. It should be noted that serializability permits concurrent execution, but ensures that concurrent actions cannot interfere with one another.

Totality solves the failure problem. Either the transfer completes entirely, in which case both the *from* and *to* accounts contain the proper new balances, or it aborts and has no effect, in which case the accounts still have their old balances.

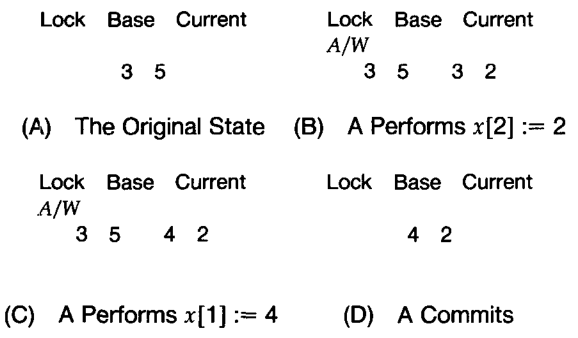
To implement serializability, we need to synchronize the accesses made by actions to shared objects. To implement totality, we need some way to recover the old state of any objects modified by an action that aborts. In Argus, synchronization and recovery are done through special objects called *atomic objects* that, like ordinary objects, provide a set of operations to access and manipulate them. However, their operations synchronize the using actions and permit the effects of aborted actions to be undone. Argus provides a number of built-in types of atomic objects, such as atomic arrays and atomic records, which have the same kinds of operations as ordinary arrays and records, but provide the additional support needed for atomicity. It also provides a mechanism for users to define new atomic data types that permit greater concurrency than the built-in atomic types. This mechanism is discussed in [20].

Synchronization for built-in atomic objects is done by means of locks. Every operation on an atomic object is classified as a reader or writer. An operation that modifies the object is a writer; other operations are readers. Readers automatically acquire a read lock on the object before accessing it; writers automatically acquire a write lock. These locks are held until the action completes, i.e., commits or aborts.[1] As is usual, there can be many concurrent holders of a read lock, but if an action holds a write lock on some object, then no other concurrent action can hold locks on that object.

Recovery is done by using versions. The state of an unlocked object is stored in a *base* version. Modifications to an object are not done to the base version directly. Instead a copy is made (in volatile memory), and modifications are done to the copy. If the action commits, the copy becomes the base version and is written to stable storage if the object is stable. If the action aborts, the copy is discarded.

An example of an atomic array is shown in Figure 4. Like all atomic objects, the array has a header with components to record the current lock holders, a base version, and, if there is an action holding a write lock, a *current* version. Figure 4A shows the object in an initial, unlocked state. Figure 4B shows what happens when the object is accessed by action A. Since A is modifying the object, a write lock is acquired on its behalf. This is possible because no other action has a lock on the object. The current version is created by copying the base version, and A's modification is made to the current version. Now A may continue to use the object. Both reading and writing are permitted since it already has a write lock. In either case, it uses the current version, so if it reads, it sees the changes it made previously. Figure 4C shows a further modification by A. Figure 4D shows what happens if A commits; the current version

[1] Thus, we are using strict two-phase locking [3].



Atomic objects implement synchronization and recovery for using actions. Locks are used for synchronization, and versions for recovery. Initially, atomic array x is unlocked and has a single base version. Then action A acquires a write lock; at this point the current version is created by copying the base version. A's modification is done to the current version and so is a subsequent modification. If A commits, the current version is installed as the base version, and the lock is discarded; if A aborts, the object reverts to its initial state.

FIGURE 4. Using Atomic Objects

replaces the base version and A's lock is discarded. If A aborts, its lock and version are discarded and the object reverts to its state in Figure 4A.

Argus allows actions to be nested [2, 17]; thus an action can have one or more subactions. Nested actions are useful for the following two reasons:

1. *They allow concurrency within an action.* An action can run many subactions in parallel. The subactions will synchronize with one another using the same rules discussed earlier. For example, all calls to get the totals of the branches in the *audit* procedure be done in parallel.
2. *They can be used to establish checkpoints within an action.* If there are several ways to accomplish a task, one can be attempted as a subaction, and, if that aborts, another can be tried without having to abort the entire computation.

Subactions require extensions to locking and version management; the complete rules are summarized in Figure 5. A subaction can acquire a read lock only if all holders of write locks are ancestors (i.e., itself, its parent, its parent's parent, and so on). It can acquire a write lock only if all holders of read or write locks are ancestors, and in this case a new version is created for its use the first time it acquires a write lock. When a subaction aborts, its locks and versions are discarded and its parent action can continue from the state at which the subaction started. If a subaction commits, its locks and versions are inherited by its parent. If the parent aborts later, all modifications of the subaction will be undone. The rules make sense because Argus does not permit a parent to run concurrently with its children, nor does it permit any concurrency within an action except by creating subactions. For example, if a parent could run concurrently with a child, then the commit of the child could overwrite changes made by the parent since the child was created. The rules are implemented by maintaining a stack of versions, one for each active action that is modifying the object. When a subaction needs a new version, the version on top of the stack is copied and the result pushed on the stack.

Argus runs every handler call as a subaction; we refer to this subaction as the *call action.* This extra action ensures that calls have a *zero* or *one* semantics: If the call is successful and the called guardian returns a reply, we guarantee that the call happened exactly once. If it is not possible to complete the call, we abort the call action, thus guaranteeing that the call (effectively) did not happen at all. Running a call as a subaction ensures that calls have a clean semantics, which is a non-trivial and desirable property in a distributed system. In addition, remote calls are often a handy place for checkpoints, since the inability to reach one guardian can sometimes be compensated for by calling a different guardian.

Also, Argus runs the processing of a handler call at the called guardian as a subaction of the call action; we refer to this subaction as the *handler action.* The handler action gives a clean separation of the calling and called guardians and ensures that each individual action runs at just one guardian. It avoids anomalies such as an action that commits at one guardian and aborts at another. It allows the handler to commit or abort unilaterally, without concern about what the calling guardian does, and similarly for the caller.

A computation in Argus starts as a *topaction,* an action that has no parent, at some guardian. The computation spreads to other guardians by means of handler calls. Execution of a handler call may cause some objects at the handler's guardian to be modified, and may in turn lead to further calls. Modifications made by these calls will be lost if a modified object's guardian

Acquiring a read lock. All holders of write locks on x must be ancestors of S.

Acquiring a write lock. All holders of read and write locks on x must be ancestors of S. If this is the first time A has acquired a write lock on x, push a copy of the object on top of the version stack.

Commit. S's parent acquires S's lock on x. If S holds a write lock on x, then S's version (which is on top of the version stack) becomes S's parent's version.

Abort. S's lock and version (if any) are discarded.

Here "ancestor" is transitive so that an action S is an ancestor of itself. Subactions can read and overwrite modifications made by ancestors, but not by unrelated actions. If a subaction commits, its parent inherits its locks and versions; if it aborts, its locks and versions are discarded.

FIGURE 5. Locking and Version Management Rules for Subaction S on Object x

crashes subsequently. When the topaction commits, it is essential that the modifications made to stable objects be written to stable storage. If this is impossible, the topaction must abort. For example, this property is needed to guarantee that a transfer would modify both (or neither) of the *from* and *to* accounts. We ensure that committing is atomic by using the two-phase commit algorithm [3] as discussed further in the section on Implementation of Action. Two-phase commit is carried out only when topactions commit.

THE BANKING SYSTEM IN ARGUS

Figure 6 shows the definition of the guardian that runs at the back end at a branch. (Argus is an extension of the CLU language [15], and most of its syntax and semantics is taken from CLU.) The guardian definition begins with a header explaining the type of guardian being defined (*branch* in this case), and the names of the operations. There are two kinds of operations. Creators are used to create new guardians of the type, while handlers are the operations provided by the guardians once they are created. For some types of guardians it is useful to have several creators, but the branch guardian has a single creator named *create*. Once a branch guardian has been created, there are five handlers that can be used to communicate with it, *open*, *close*, *deposit*, *withdraw*, and *total*.

The first definitions to appear inside a guardian definition are type definitions and declarations of the variables that make up the guardian's state. In this case, the entire state of the guardian is stable, since all the variables making up the state are declared to be stable. The state consists of three objects: a hash table providing access to accounts, the unique code for this branch, and the seed used to generate unique names for new accounts.

Since deposits and withdrawals are likely to happen frequently, we want them to be fast. There are several issues to consider here. First, locating the account of interest must be fast. Second, concurrent deposits and withdrawals are quite likely, so we want to allow them when possible. Finally, we want to minimize the amount of writing to stable storage needed to record the results of the various operations.

The representation used in the branch guardian achieves these goals. To find the account, we use the hash table, *ht*, which maps from integers (obtained from hashing the account number) to buckets. The name *htable* is used as an abbreviation for *ht's* type (atomic_ array[bucket]). A bucket is an atomic array, each of whose elements contains information about an account that hashes to that bucket. (The name *bucket* is used as an abbreviation for this type.) The information stored is the account number and the object that records the information about the account itself. An account number is an atomic record with two components; the first component records the code for its branch, while the second is an integer that is unique for its branch. The only information stored for an account is its balance; in a real implementation, of course, much more information would be stored. The seed is stored as an atomic_ record with a single integer component to ensure proper synchronization of concurrent *opens*. It should be noted that all the data structures used are atomic objects, which means that actions using the branch guardian will be synchronized properly and that their effects will be undone if they abort. Read and write locks on atomic objects are mostly acquired automatically when operations are invoked; e.g., the last statement of *open* acquires a read lock on *ht* (since it reads it to obtain the bucket of the new account) and a write lock on the bucket, since it modifies it to add the new account.

Since there is no volatile state for this guardian, there is no need for any code to run during crash recovery. Furthermore, the guardian has no background code; it does all its work as part of carrying out the handler calls. Therefore, the remainder of the guardian consists of definitions of the creator and handlers, plus an internal procedure, *lookup*.

The creator, *create*, takes the branch's code and the hash table size as arguments. It initializes the guardian state and then returns itself, i.e., the newly created guardian. The hash table is initialized to contain a full complement of empty buckets. The buckets can be empty initially because arrays in Argus grow (and shrink) dynamically. In the code of *create*, there are several illustrations of the notation used in Argus to name operations. For example, the notation *htable$new* names the *new* operation of the *htable* type.

Total computes the sum of the balances of the accounts at the branch by using *iterators* [15]. An iterator is a special kind of operation that yields its results instead of returning. When the iterator yields, its result is assigned to the loop variable and the loop body is run. When the body completes, control resumes in the iterator so it can produce the next result; and when the iterator has no more results to yield, both it and the loop terminate. The *elements* iterator used here produces all elements of the array from the first to the last. *Total* uses nested iterators. The iterator in the outer **for** statement produces every bucket in the hash table. Each account in the bucket is accessed in the inner **for** statement.

When *total* finishes, it commits and returns the computed sum. As mentioned, a handler runs as a subaction of the calling action. When a handler terminates, it can either commit or abort. The default is committing; in the absence of an explicit command to abort, the handler action will commit. An explicit abort is indicated by prefixing a **return** or **signal** with **abort**. All handlers in the example terminate by committing; in our experience, this is the most common case by far. *Total* acquires a read lock on the hash table and also on each bucket since the iterators read the hash table and the buckets. It also acquires a read lock on each account when it reads the balance. These locks are acquired by its parent when it commits.

Open generates an account number for the new account and advances the seed. It obtains a write lock on

This guardian implements the database for one branch. It maintains information about accounts in stable storage, and provides operations to open and close accounts, to withdraw and deposit money in accounts, and to provide the total of all accounts in the database.

FIGURE 6. The Branch Guardian

```
branch = guardian is create handles total, open, close, deposit, withdraw

  % type definitions
  htable = atomic_array[bucket]
  bucket = atomic_array[pair]
  pair = atomic_record[num: account_number, acct: acct_info]
  acct_info = atomic_record[bal: int]
  account_number = atomic_record[code: string, num: int]
  intcell = atomic_record[val: int]

  stable ht: htable          % the table of accounts
  stable code: string        % the code for the branch
  stable seed: intcell       % the seed for generating new account numbers

  create = creator (c: string, size: int) returns (branch)
    code := c
    seed.val := 0
    ht := htable$new( )
    for i: int in int$from_to(1, size) do
      htable$addh(ht, bucket$new( ))
      end
    return (self)
    end create

  total = handler ( ) returns (int)
    sum: int := 0
    for b: bucket in htable$elements(ht) do
      for p: pair in bucket$elements(b) do
        sum := sum + p.acct.bal
        end
      end
    return (sum)
    end total

  open = handler ( ) returns (account_number)
    intcell$write_lock(seed) % get a write lock on the seed
    a: account_number := account_number${code: code, num: seed.val}
    seed.val := seed.val + 1
    bucket$addh(ht[hash(a.num)], pair${num: a, acct: acct_info${bal: 0}})
    return (a)
    end open

  close = handler (a: account_number) signals (no_such_acct, positive_balance)
    b: bucket := ht[hash(a.num)]
    for i: int in bucket$indexes(b) do
      if b[i].num ~= a then continue end
      if b[i].acct.bal > 0 then signal positive_balance end
      b[i] := bucket$top(b) % store topmost element in place of closed account
      bucket$remh(b) % discard topmost element
      return
      end
    signal no_such_acct
    end close

  lookup = proc (a: account_number) returns (acct_info) signals (no_such_acct)
    for p: pair in bucket$elements(ht[hash(a.num)]) do
      if p.num = a then return (p.acct) end
      end
    signal no_such_acct
    end lookup

  deposit = handler (a: account_number, amt: int) signals (no_such_acct, negative_amount)
    if amt < 0 then signal negative_amount end
    ainfo: acct_info := lookup(a) resignal no_such_acct
    ainfo.bal := ainfo.bal + amt
    end deposit

  withdraw = handler (a: account_number, amt: int)
          signals (no_such_acct, negative_amount, insufficient_funds)
    if amt < 0 then signal negative_amount end
    ainfo: acct_info := lookup(a) resignal no_such_acct
    if ainfo.bal < amt then signal insufficient_funds end
    ainfo.bal := ainfo.bal - amt
    end withdraw

  end branch
```

the seed first to prevent deadlocks between concurrent *opens*. (The deadlock would occur if two *opens* each obtained a read lock on the seed; then neither would be able to obtain the write lock needed to increment the seed.) *Open* uses the *hash* procedure to compute the bucket of the new account (the code of this procedure is not shown in the example). *Hash* takes in the integer part of the account number, it returns an integer between zero and the current size of the hash table. *Open* uses the array *addh* operation, which extends the array by one and stores the element passed to it as an argument in the new position, to enter the new account in the accounts table.

Close looks up the account by using the array *indexes* iterator to search the account's bucket; this iterator returns all the legal indexes in the array. The other handlers make use of the internal procedure, *lookup*, to find the entry in the map for a particular account, or signal *no_match* if there is no such account.

The implementation provides lots of concurrent activity. Concurrent deposits and withdrawals are permitted on different accounts. This concurrency is allowed because these operations acquire only read locks on the hash table and on the account's bucket. A write lock is acquired only on the information stored for the account itself.

Close can run in parallel with calls of *open*, *close*, *deposit*, and *withdraw*, provided those other calls make use of different buckets. It prevents other calls that use the same bucket because it acquires a write lock on the bucket (when it assigns to the i^{th} element of the bucket). *Open* is similar to *close* except that it also excludes other concurrent *opens* because each *open* acquires a write lock on the *seed*. This exclusion is not a problem if buckets are small and if accounts are opened rarely. If it is a problem, the state of the guardian can be implemented differently, possibly using mechanisms for user-defined atomic types, to reduce conflicts. For example, user-defined types could permit concurrent *opens*.

The amount of writing to stable storage is small in all cases. For *deposit* and *withdraw*, only the account itself must be written to stable storage. For *open* and *close*, the bucket of the opened or closed account must be written. The accounts contained by the buckets are not written, except for the newly opened account. Nothing need be written for *total* since it does not modify anything.

The implementation of *total* is the main problem with the guardian. *Total* is slow because it needs to examine the balances of all accounts. It could be improved by keeping a running total, but then deposits and withdrawals would conflict with one another because each would need to change the total and thus would require a write lock. A user-defined atomic type would permit us to keep a running total without having conflicts between deposits and withdrawals.

In addition, *total* conflicts with all other operations: until the topaction that called *total* completes, other operations at that branch are delayed. Conflicts with deposits and withdrawals are necessary if the reported total is to be up to date. They could be avoided by having *total* return a sum that is slightly out of date. Again, a user-defined atomic type would help here.

As implemented, most of the handlers can deadlock with other concurrent operations. For example, *deposit* can deadlock with other *deposits* or *withdrawals* on its account. The reason for the deadlock is that the operation first obtains a read lock on the account and then later needs a write lock. Such a problem was avoided in the implementation of *open* by obtaining the write lock first; a similar solution can be used here.

No operations modify the hash table. This is important for performance since an operation that modified the hash table would conflict with all other operations. Also, the hash table is big, so we do not want to copy it to stable storage. Of course, it might be necessary to reorganize the guardian by changing the size of the hash table, or using a different method of hashing. Such a change could be installed by running a topaction, and copying the new hash table and buckets to stable storage when that topaction commits.

The Front-end Guardian

A portion of the *front-end* guardian is shown in Figure 7. This guardian has no handlers; all its work is done in the background code. When it is created, it passes information about the devices it is controlling, and also the identity of another guardian (of type *registry*) whose job is to remember how branch codes are related to branch guardians. This guardian provides various operations to access its information. The device information and the identity of the registry guardian are kept in stable storage. To speed up processing of requests, the front end maintains in table *bt* a volatile copy of the information stored by the registry guardian. This copy is initialized by the creator and also by the recovery code after a crash.

A transfer is carried out by interacting with the user to determine the *from* and *to* accounts and the *amount*. A topaction is then created and the transfer is carried out within it. *Get_branch* is used to determine the branches of the two accounts; it extracts the code of an account and looks it up in the *bt* table. The calls to the two branches are done in parallel, each in its own subaction. If both calls return normally, the **coenter** completes, committing both its subactions, and then the topaction commits. If either call signals an exception, the **coenter** is halted immediately, aborting the other call if it has not yet completed, and then the topaction aborts. The topaction must be aborted in this case because it is possible that the call in other arm (the one that did not raise an exception) terminated first, in which case the transfer is partly done. Aborting the topaction will undo the effects of the other call in this case.

To carry out an audit, the background code interacts with its user to determine what branches are of interest, and then runs the entire audit as a topaction. It communicates with all branches in parallel, using a separate process and subaction for each call. As each call returns, the total is incremented. The total is maintained in an atomic record to ensure proper synchronization of the accesses in the arms. Each arm obtains a write lock on the total first to avoid deadlocks with other arms that are running concurrently. If all calls return normally, the topaction commits.

Even though the *total* handler signals no exceptions, it is still possible for its call to terminate with an exception. This can happen, for example, because it is impossible to communicate with the handler's guardian, even after repeated tries. In such a case, the Argus system will terminate the call automatically with the *unavailable* exception. If such an exception occurs, the **coenter** is terminated immediately, aborting any unfinished arms, and then the topaction aborts.

It is important for actions to be short since they hold locks and therefore can interfere with other actions. For this reason, all communication with the user is done outside of actions for both transfers and audits. A crash before the front end informs the user of the outcome of a request can leave the user uncertain about whether the request completed. All *external* actions, i.e., those that interact with the external environment, have this problem. The problem cannot be solved by moving the interaction inside the action, because then the action might not commit after telling the user the transfer had completed.

The system as shown so far is static: e.g., no provision is made for adding new branches. The registry guardian can be used to support dynamic reconfiguration of the system. The front end could carry out a dialog with the user and then interact with the registry to enter the new information. For example, to add a new branch, the user could define an appropriate code for the branch, or the registry could do this. The user would need to indicate where the new branch guardian should reside; Argus provides a built-in datatype, *node*, for this purpose. The registry would then create the new guardian by the statement:

```
b: branch := branch$create(c) @ n
```

This will create a new branch at the node indicated by *n*, and then run the creator in this new guardian, passing it code *c* as an argument. The new guardian returned by *create* can then be stored in the registry's tables.

To run in such a dynamic system, the front end must be prepared for information in its *bt* table to be out of date. For example, when the *get_branch* procedure looks up an account number, it might discover an unknown code. In this case, it would read the table from the registry and try again. Having only a single registry would be an availability bottleneck in the dynamic system and can be avoided by replicating the registry.

IMPLEMENTATION OF ACTIONS

The current implementation of Argus is a prototype running on MicroVAX-IIs under Unix version Ultrix 1.2. The machines communicate over a 10 megabit/second ethernet. Each MicroVax has either 9 or 13 megabytes of primary memory and two RD53 disks, each with 70 megabytes of disk storage. One of the disks is used for our stable storage.

The implementation was done with limited manpower, and we have not optimized it the way we would if it were a production system. For example, for the most part we have avoided making modifications to the Unix kernel. Nevertheless, we designed the implementation carefully and thus avoided certain pitfalls. An area of particular interest is the way we implement actions, because this is where Argus differs most from other implemented systems. Our experience indicates that nested actions do not require significant overhead. Top-level actions do have a cost, but this cost can be minimized by careful design. We will now sketch our implementation of actions and give some data on the performance of our system; a more thorough discussion appears in [12].

Our implementation of actions is designed to avoid unnecessary delay of user computations. There are two main forms of delay to avoid: extra communication, and writes to stable storage. In general, communication delays are avoided by piggybacking information on messages that must be exchanged anyway, and by communicating information in background mode. When multiple guardians participate in an action, delay is minimized by performing stable storage writes concurrently at all guardians. In addition, most writing to stable storage can take place in background mode. One place where delays cannot be avoided is when topactions commit; at this point it is necessary to communicate with the guardians where descendants ran, and some writing to stable storage is required.

The activity that takes place when various events occur, such as creating and terminating actions, is summarized in Table I. Creating top- and subactions is done locally at the guardian where the subaction runs. (Recall that each action runs entirely at a single guardian.) All that is needed is to create a unique identifier for the new action and to initialize some data structures associated with it. For example, we keep track of all guardians visited by committed descendants of an action in the *plist*, which becomes the list of participants that is used during two-phase commit. Initially, the *plist* for a sub- or topaction contains one guardian, the creating guardian.

When a handler subaction commits, its guardian remembers the local atomic objects it has locked. (As mentioned earlier, a handler subaction is created for running the processing of a handler call.) The reply message indicates that the action committed and also contains other information such as the subaction's *plist*. The guardians in its *plist* are added to its parent's *plist* when the reply message arrives.

```
frontend = guardian is create

    btable = atomic_array[binfo]
    binfo = atomic_record[code: string, branch: branch]
    account_number = atomic_record[code: string, num: int]
    intcell = atomic_record[val: int]

    stable central: registry  % maintains relationship between codes and branches
    stable dev: device_info
    bt: btable  % relates codes to branches; this is volatile

    recover
        bt := central.get_branch_info( )
        end

    background
        % the background code listens to the various devices using a separate process for each one
        % and carries out commands of its users

        % for a transfer it does the following
        % find out accounts and amounts from user and store in local variables to, from and amt
        enter topaction
            t: branch := get_branch(to)
            f: branch := get_branch(from)
            coenter
                action f.withdraw(from, amt)
                action t.deposit(to, amt)
                end except others: abort exit problem end  % all exceptions cause abort of topaction
            end  %  topaction
                except when problem: % tell user that transfer failed
                    end  % except
        % tell user that transfer succeeded

        % for an audit it does the following:
        blist: array[branch] := % put in branches of interest to user
        total: intcell := intcell${val: 0}  % initialize total
        enter topaction
            coenter
                action foreach b: branch in array[branch]$elements(blist)
                    t: int := b.total( )
                    intcell$write_lock(total)
                    total.val := total.val + t
                end except others: abort exit problem end  % all exceptions cause abort
            end except when problem:  % tell user that audit cannot be done now
                    end
        % tell user the result

        end % background

    create = creator (c: registry, d: device_info) returns (frontend)
        central := c
        dev := d
        bt := central.get_branch_info( )  % get information about branches from central
        return (self)
        end create

    get_branch = proc (a: account_number) returns (branch) signals (no_such_account)
        for b: binfo in btable$elements(bt) do
            if b.code = a.code then return (b.branch) end
            end
        signal no_such_account
        end get_branch

end frontend
```

The front-end guardian has no handlers. It interacts with clerks and other users, and also with cash vending machines, in the background code. It makes handler calls to branch guardians to carry out requests and uses topactions to ensure that requests happen atomically. The processing of transfers and audits is shown.

FIGURE 7. A Portion of the Front-end Guardian

When any other subaction commits, its locks and versions for objects belonging to its guardian are propagated to its parent, but locks and versions for objects belonging to other guardians are not propagated, since this would require communication. (It would have acquired these locks and versions from its committed handler action descendants.) Similarly, aborting a topaction or a subaction releases its locks and versions for local objects but not for nonlocal objects. In addition, in this case we send abort messages to the appropriate guardians in background mode. We do not guarantee that such messages will arrive, although they do arrive with high probability.

Since abort messages may not arrive, and messages about commits of subaction ancestors are not even sent, the guardian where the locked object resides may have out-of-date information. For example, an object it has marked as locked may actually be unlocked. To determine the true state of such an object, it can send a query message to some other guardian. Query messages are directed to guardians where ancestors of the action currently holding the lock ran. They are typically sent when some other action needs the lock. They relieve us of the need to guarantee delivery of abort messages and of the need to send any messages about commits of ancestors.

TABLE I. Running Actions

Event	Activity
Creating an action	Create action identifier, and initialize action state; all work is local to the action's guardian
Aborting a top- or subaction	Discard locks and versions locally; send abort messages to guardians of committed descendants in background
Committing a subaction	Propagate locks and versions locally
Committing a topaction	Carry out two-phase commit unless the action has no nonlocal descendants

Most action events require only local processing. Each action runs at a single guardian. That guardian creates it (by creating its identifier and initializing some associated state) and handles its termination. Committing or aborting a subaction, or aborting a topaction, is done entirely at the guardian, although, if the action aborts, abort messages are sent in background to notify other guardians. Commits of topactions require the two-phase commit protocol to be carried out unless the topaction has no nonlocal committed descendants.

When a topaction commits, the system carries out the two-phase commit protocol [3] to ensure the action either commits everywhere or aborts everywhere. The participants in the protocol are the guardians in the *plist*; the coordinator is the topaction's guardian. In Phase One the coordinator sends prepare messages to all participants. Each participant records the versions written by (descendants of) the preparing action on stable storage, writes a *prepare* record to stable storage, and then responds "ok." (To speed up two-phase commit, versions can be written to stable storage while in background mode at the participants, although this optimization has not yet been implemented.) The participant also releases all read locks held by the action at this point. If the participant is unable to record the necessary information because it crashed after the subaction ran at it, it responds "refused."

If all participants respond "ok," then the coordinator writes a *committed* record to stable storage and enters Phase Two by notifying all participants to commit the action. (We optimize to avoid Phase Two for read-only actions.) The rest of this phase is carried out in background mode. When a participant receives a "commit" message, it records the commit on stable storage, installs the action's versions and releases its locks, and then notifies the coordinator that it is done. The coordinator continues to re-send commit messages to participants until it gets this acknowledgement. Thus we guarantee that commit messages are delivered eventually; the *committed* record contains the *plist* so that this promise can be kept even if the coordinator crashes. When all participants acknowledge, the coordinator records this fact on stable storage, and the second phase is over.

If some participant responds "refused" or does not respond, the coordinator aborts the transaction and sends abort messages to the participants. When a participant receives an abort message, it records it on stable storage and discards the action's versions and locks. The abort messages are sent in background mode and we do not guarantee delivery. As discussed earlier, participants can send query messages to recover from lost abort messages.

The minimal delay of user code for committing a topaction is effectively one network round-trip (the "prepare" and "ok" messages) plus two writes to stable storage (the *prepare* and *committed* records). The minimal delay occurs when all versions at participants have already been written to stable storage before the prepare message arrives.

Our method of implementing actions has proved quite satisfactory. Piggybacking action information in messages and being lazy about propagating information about aborts and commits of subactions are both good ideas. Subactions are cheap as a result. The use of queries as a backup mechanism is also good because it eliminates the need for reliable communication in many cases. Queries place the responsibility for making sure information is communicated on the guardian that needs to know what happened. As a result, other guardians need not remember events such as aborts of actions.

Some performance information is given in Table II, although a more thorough analysis can be found in [12]. All data in the figure are in milliseconds. The data show that our costs are dominated by communication and disk writes. Committing and aborting subactions, and also committing a local, read-only topaction, are inexpensive because neither communication nor writing to disk is required. Committing a local topaction that modified an object requires writing to disk but not two-phase commit.[2] A read-only topaction with one participant does one handler call to a remote guardian, and then later does Phase One of two-phase commit; no writing to disk is required. (Handler calls take approximately 17.5 milliseconds.) An updating topaction requires writing to disk and both phases of two-phase commit.

TABLE II. Data on Action Commits and Aborts

	Read-only	Update
Subaction commits	0.60	0.81
Subaction aborts	0.65	0.85
Local topaction commits	0.63	17.50
Topaction with one participant commits	36.50	82.00

All data are in milliseconds. Subactions are cheap because all processing is local. Local topaction commit requires no communication; for a read-only action, the cost is equivalent to a read-only subaction. For an update action, however, one record must be written to stable storage. A topaction with a participant makes a handler call and later does communication during two-phase commit. Only one phase of communication is needed for the read-only transaction. For the updating transaction, two phases of communication and four writes are needed, although most of the second phase, including the second pair of writes, takes place in background mode.

Our current implementation is a prototype that was developed primarily to test the soundness of our ideas and to provide a testbed for experimentation, and many obvious optimizations have not been done. Under these circumstances, we consider our performance quite satisfactory and believe it indicates the practicality of systems like ours. These conclusions are borne out by data from other projects such as Camelot [19].

CONCLUSIONS

Argus is intended to support distributed implementations of systems that maintain on-line state for users. Guardians can be used to control where data and processing are located, and they are resilient, so information is not lost in crashes. Also, they support dynamic reconfiguration since they can be created, moved, and destroyed dynamically and handler calls are location independent. Atomic actions allow online information to be maintained consistently in spite of failures and concurrency and make it relatively easy to improve system availability by replicating information.

Argus is unique because it provides atomic actions within a programming language. Several other languages, e.g., SR [1], Ada [8], and Mesa [16], support distributed computing but not transactions. In addition, a new language [7] is being developed that is similar to Argus. Transactions arose in database systems [3], where they are made available to users via the data base but not for general objects as in Argus. There are also a number of systems that provide operating system support for actions but not language support [18, 19].

[2] A raw disk write without a seek requires approximately 17 ms; we do not implement true stable storage (which requires two sequential writes) at present.

Argus has been running for about two years, although early in this period the implementation was quite incomplete. We have used it to implement a number of distributed programs, including the following:

1. A preliminary version of a library in which information about programs is stored. The library allows Argus guardians and other modules to be developed by different people at different locations while still enforcing compile-time type checking of module interfaces. Also, it provides stable storage for program code.
2. The catalog that allows run-time lookup of guardians and handlers. For example, a program could use the catalog to find a printer spooler for a printing device.
3. A distributed editor [5] that allows users at different locations to collaborate on the same document.
4. A mail repository, which provides mailboxes for storing and retrieving mail for users. The mail repository uses replication to provide high availability, and is also designed to permit a wide range of reconfigurations.
5. A program to compute Hailstone numbers (see [6]).
6. A distributed game that allows users at different machines to take part in the same game.

Our Argus experience to date indicates that it is relatively easy to write distributed programs, even sophisticated ones that replicate information to increase availability and that are able to reconfigure themselves dynamically. The system is helpful because atomic actions are an important tool both for understanding what a system should do and for implementing it correctly.

It is important to understand that atomic actions do not constrain the kinds of systems that can be built. It is up to the person defining a system to decide how quickly effects of computations must propagate to other parts of the system, and to decide how much information can be lost in case of a crash. Actions can be used to implement a spectrum of requirements. If quick propagation and no lost information are required, the cost of actions will be greater than what is needed to support weaker requirements.

For example, in the banking system shown earlier, deposits and withdrawals are visible to other users of the system as soon as their topaction commits. Suppose instead that the designer of the banking system decides that deposits should have a delayed effect. This could be implemented by having the front end (see Figure 7) record the deposit in local stable storage and complete it at night, for instance.

Another example of weak constraints is the program that computes Hailstone numbers [6]. This program uses different guardians to do parallel searches for numbers in separate ranges. It has a stable state so that the results of past computations will not be entirely lost in crashes. It uses atomic actions to take periodic checkpoints and to coordinate interactions between the searching guardians and a front end that is responsible for assigning ranges to guardians. Checkpoint actions run every 10 minutes; coordination actions about once a day. Actions made these parts of the program very easy to implement, yet their impact on performance is negligible.

Argus does not free the programmer from concern with details of concurrency. The programmer must think about deadlocks and starvation and implement the code to avoid them when possible. Often deadlocks are program errors, but this is not always true, e.g., a concurrent transfer and audit could deadlock. The Argus implementation does not detect deadlocks at present, but we intend to add such detection capabilities shortly.

Programmers must also think carefully about the efficiency of data representations. Both the amount of writing to stable storage and the degree of concurrency are of interest. User-defined atomic types can be used where necessary to improve performance and remove some deadlocks. User-defined atomic types are complicated to implement, but are not needed very often.

Although we are quite happy with Argus, there are areas where we think more work is needed. Our user-defined atomic type mechanism, for example, is more complicated than we would like. We need better support for reconfiguration than is available at present. Also, sometimes it would be useful for a guardian to explicitly run code when an ancestor of some action that ran there commits or aborts.

For now, however, we plan to leave Argus essentially unchanged and to use it to build a number of additional applications, such as an object repository. Some of these applications will be used by a large community of users. To run them effectively, we intend to improve the performance of the implementation in some areas. Communications is one area where we believe we can reduce our costs substantially by implementing some special Unix kernel calls. We will do a full evaluation of Argus when these applications are complete and in use.

Acknowledgments. Argus is the result of a collaboration with a number of other researchers, including Maurice Herlihy, Paul Johnson, Robert Scheifler, and William Weihl. The author is grateful to all those who helped develop Argus, and also to the referees of earlier drafts of this article.

REFERENCES

1. Andrews, G.R., and Olsson, R.A. The evolution of the SR language. *Distrib. Comput. 1*, 2 (Apr. 1986). Also Tech. Rep. 85-22, Univ. of Arizona, Tucson, Ariz., Oct. 1985.
2. Davies, C.T. Data processing spheres of control. *IBM Syst. J. 12*, 2 (1978), 179–198.
3. Gray, J.N. Notes on data base operating systems. In *Lecture Notes in Computer Science*. G. Goos and J. Hartmanis, Springer-Verlag, New York, 1978, pp. 393–481.
4. Gray, J.N., Lorie, R.A., Putzolu, G.F., and Traiger, I.L. Granularity of locks and degrees of consistency in a shared data base. In *Modeling in Data Base Management Systems*. G.M. Nijssen, Ed. North Holland, Amsterdam, 1976.
5. Greif, I., Seliger, R., and Weihl, W. *A case study of CES: A distributed collaborative editing system implemented in Argus.* Programming Methodology Group Memo 55. MIT Laboratory for Computer Science, Cambridge, Mass. Apr. 1987. To be published in IEEE Transactions on Software Engineering.
6. Hayes, B. Computer recreations: On the ups and downs of hailstone numbers. *Scientif. Amer. 250*, 1 (Jan. 1984).
7. Herlihy, M., and Wing, J. Avalon: Language support for reliable distributed systems. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, (Pittsburgh, Pa., July). IEEE, New York, 1987.
8. Ichbiah, J., et al. Rationale for the design of the Ada programming language. *SIGPLAN Not. 14*, 6 (June 1979).
9. Lampson, B.W., and Sturgis, H.E. *Crash recovery in a distributed data storage system.* Tech. Rep. Xerox Research Center, Palo Alto, Calif., 1979.
10. Liskov, B. *Overview of the Argus language and system.* Programming Methodology Group Memo 40. M.I.T. Laboratory for Computer Science, Cambridge, Mass., Feb. 1984.
11. Liskov, B., et al. *Argus reference manual.* Tech. Rep. MIT/LCS/TR-400. M.I.T. Laboratory for Computer Science, Cambridge, Mass., 1987.
12. Liskov, B., Curtis, D., Johnson, P., and Scheifler, R. Implementation of Argus. In *Proceedings of the 11th Symposium on Operating Systems Principles* (Austin, Texas, Nov.). ACM, New York, 1987.
13. Liskov, B., and Guttag, J. Iteration abstraction. In *Abstraction and Specification in Program Development*. MIT Press, Cambridge, Mass., and McGraw Hill, New York, 1986.
14. Liskov, B., and Scheifler, R.W. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Prog. Lang. Syst. 5*, 3 (July 1983). 381–404.
15. Liskov, B., Snyder, A., Atkinson, R.R., and Schaffert, J.C. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug. 1977). 564–576.
16. Mitchell, J.G., Maybury, W., and Sweet, R. *Mesa language manual version 5.0.* Tech. Rep. CSL-79-3. Xerox Research Center, Palo Alto, Calif., 1979.
17. Moss, J.E.B. *Nested transactions: An approach to reliable distributed computing.* MIT Press, Cambridge, Mass., 1985.
18. Mueller, E., Moore, J., and Popek, G. A nested transaction mechanism for LOCUS. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles* (Bretton Woods, N.H., Oct.). ACM, New York, 1983.
19. Spector, A.Z., et al. *Camelot: A distributed transaction facility for Mach and the Internet—An interim report.* Tech. Rep. CMU-CS-87-129. Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., 1987.
20. Weihl, W., and Liskov, B. Implementation of resilient, atomic data types. *ACM Trans. Prog. Lang. Syst. 7*, 2 (Apr. 1985), 244–269.

CR Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Performance of Systems**]: Reliability, Availability, and Serviceability; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.2 [**Programming Languages**]: Language Classifications—*very high level languages*; *Argus*; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types*; *concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; *deadlocks*; *synchronization*; D.4.4 [**Operating Systems**]: Communications Management—*network communication*; D.4.5 [**Operating Systems**]: Reliability; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; H.2.4 [**Database Management**]: Systems

General Terms: Languages, Reliability

Author's Present Address: Barbara Liskov, MIT, Laboratory for Computer Sciences, 545 Technology Square, Cambridge, MA 02139.