FT-SR: A Programming Language for Constructing Fault-Tolerant Distributed Systems

Richard D. Schlichting Vicraj T. Thomas

TR 92-31

FT-SR: A Programming Language for Constructing Fault-Tolerant Distributed Systems [1]

Richard D. Schlichting Vicraj T. Thomas

TR 92-31[2]

Abstract

The design and implementation of FT-SR, a programming language oriented towards constructing fault-tolerant distributed systems, is described. The language, which is based on the existing SR language, is unique in that it has been designed to support equally well any of the programming paradigms that have been developed for this type of system, including the object/action model, the restartable action paradigm, and the state machine approach. To do this, the language is designed to support the implementation of systems modeled as collections of fail-stop atomic objects. Such objects execute operations as atomic actions except when a failure or series of failures cause underlying implementation assumptions to be violated; in this case, notification is provided. It is argued that this model forms a common link among the various paradigms and hence, is a realistic basis for a language designed to support the construction of systems that use any or all of these approaches. An example program consisting of a data manager and its associated stable storage is given; the manager is built using the restartable action paradigm, while the stable storage is structured using the replicated state machine approach. Finally, an implementation of the language that uses the x-kernel and runs standalone on a network of Sun workstations is discussed.

November 25, 1992

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This work supported in part by the National Science Foundation under grant CCR-9003161 and the Office of Naval Research under grant N00014-91-J-1015.

[2]This report supersedes TR 91-24a

1 Introduction

Ensuring the *dependability* of computer systems—that is, that the system delivers services on which people can rely [Lap91]—is an increasingly important issue. One key aspect of this problem is the development of techniques and support systems for constructing *fault-tolerant distributed programs* that can continue to execute despite the failure of one or more processors in a distributed system. Such programs are intimately tied to the problem of increasing dependability, since many systems in a variety of areas ranging from databases to process control are, in fact, fault-tolerant distributed programs of one kind or another.

Constructing programs of this type is undeniably difficult, which has led to research in a variety of areas aimed at systematizing and simplifying various aspects of the task. For instance, *failure models* have been developed as a means for precisely specifying assumptions made about the possible effect of failures. Examples of popular failure models include fail-stop [SS83], timing [Cri91], fail-silent [PVB+88], and Byzantine [LSP82]. Another area has been the development of *programming paradigms*, which simplify the development of certain types of fault-tolerant distributed programs by providing canonical organization techniques and abstractions. Examples of popular programming paradigms include the object/action model [Gra86], the restartable action paradigm [Lam81], and the replicated state machine approach [Sch90].

In this paper, we focus on a third area related to simplifying the construction of fault-tolerant distributed programs, that of providing adequate programming language support. Specifically, we describe the design and implementation of FT-SR, a programming language based on SR [AOC+88, AO93] that is oriented towards writing fault-tolerant distributed systems. FT-SR is unique in that it has been designed to be a *multi-paradigm language*, that is, a language that can support equally well *any* of the multiple programming paradigms that have been developed for this type of system. This is done by providing support for constructing *fail-stop atomic objects*; such objects behave as atomic objects unless their implementation assumptions are violated, in which case notification is provided. Here, we concentrate on processors with fail-silent semantics—that is, where the only failures are assumed to be a complete cessation of execution activity—although the approach generalizes to other failure models as well. The language has been implemented using the x-kernel, an operating system designed for experimenting with communication protocols [HP91], and runs standalone on a network of Sun workstations.

The orientation towards supporting multiple paradigms distinguishes FT-SR from other languages [Lis85, EFH82, LW85], language extensions [SCP91, CGR88, KU87] and language libraries [BSS91, Coo85, PS88, HW87] related to fault-tolerance, which are typically oriented around a particular paradigm. Support for a single paradigm has been shown to be constraining in many situations [Bal91], and is particularly inappropriate for constructing systems, where different paradigms may be used at different levels of abstraction. Moreover, the development of such a multi-paradigm language for fault-tolerant programming can be viewed as analogous to the evolution of standard distributed programming languages, which have progressed from languages such as CSP [Hoa78] and Concurrent Pascal [BH75] that support only a single synchronization paradigm to those such as SR, Dislang [LL81], Pascal-FC [BD88], and StarMod [Coo80] that support multiple approaches.

This paper is organized as follows. In Section 2, we first describe fail-stop atomic objects and the programming model that results from considering these as the primary abstraction. We also argue that this model is a "lowest common denominator" for the various programming paradigms, and hence, a realistic basis for the design of a language intended to support these approaches. Section 3 then outlines the design of FT-SR and describes how its features facilitate the implementation of fail-stop atomic objects using commonly accepted techniques. The use of the language is illustrated in Section 4 with the presentation of a data manager and associated stable storage. Section 5 provides an overview of the implementation, focusing particularly on the language runtime where the inclusion of new features has had the biggest impact. Section 6 returns to the issue of the appropriateness of FT-SR's mechanisms, and elaborates further on related work. Finally, Section 7 offers some conclusions.

2 Fail-Stop Atomic Objects

As mentioned, our programming model is based on the abstraction of a fail-stop (or FS) atomic object. Such an object contains one or more threads of execution, which implement a collection of operations that are exported and made available for invocation by other FS atomic objects. Higher-level FS atomic objects may, in turn, be constructed by composition. When invoked, an operation exported by an FS atomic object normally executes as an atomic action that is both *unitary*—all or nothing despite failures—and *serializable*—executed relative to other atomic actions such that the result is equivalent to some serial schedule [Lam81]. However, as is always the case with fault-tolerance, these properties can only be *approximated* by an implementation; that is, they can be only guaranteed *relative* to some set of assumptions concerning the number and type of failures. For example, algorithms to realize the unitary property often rely on stable storage in such a way that the failure of this abstraction can lead to unpredictable results. Or, a series of untimely failures might exhaust the redundancy of an implementation built using replication.

To account for cases such as these, the semantics of FS atomic objects include the concept of *failure notification*. Such a notification is generated for a particular object whenever a *catastrophic failure* occurs, where such a failure is defined to occur when an object's implementation assumptions are violated, or should the object be explicitly destroyed from within the program. The status of an operation being executed when such a failure notification occurs is indeterminate. Hence, the analogy to fail-stop processors implied by the term "fail-stop atomic objects" is strong: in both cases, either the abstraction is maintained (processor or atomic object) or notification is provided.

A fault-tolerant distributed system can be realized by a collection of FS atomic objects organized along the lines of functional dependencies. For example, an FS atomic object implementing the services of a transaction manager may use the operations exported by another FS atomic object implementing the abstraction of stable storage [Lam81]. These dependencies can be defined more formally using the *depends* relation given in [Cri91]. In particular, an FS atomic object u is said to *depend* on another object v if the correctness of u's behavior depends on the correctness of v's behavior. Thus, the failure of v may result in the failure of u, which in turn can lead to the failure of other objects that depend on u.

Increasing the dependability of a distributed system organized in this way is done by decreasing the probability of failure of its constituent FS atomic objects using fault-tolerance techniques based on the exploitation of redundancy. For example, an object can be replicated to create a new FS atomic object with greater resilience to failures. This replication can either be active, where the states of all replicas remain consistent, or passive, where one replica is a primary and others remain quiescent until a failure occurs. Or, an FS atomic object can contain a recovery protocol that would be executed upon restart following a failure to complete the state transformation that was in effect when the failure occurred. The applicability of each of these techniques depends on the details of the system or the application being implemented.

As an example of how a typical fault-tolerant system might be structured using FS atomic objects, consider the simple distributed banking system shown in Figure 1. Each box represents an FS atomic

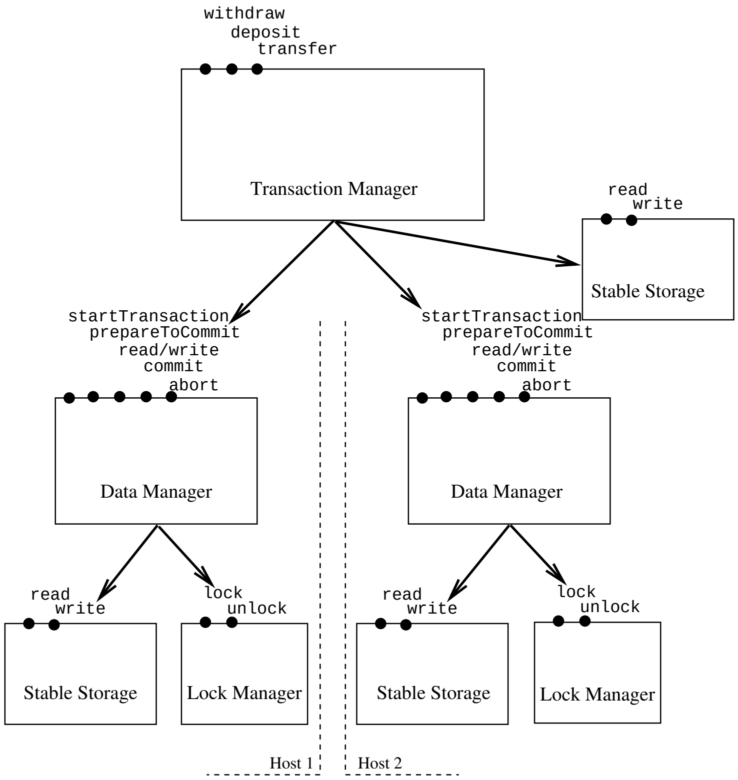

Figure 1: Fault-tolerant system structured using FS atomic objects

object, with the dependencies between objects represented by arrows. User accounts are assumed to be partitioned across two processors, with each data manager object managing the collection of accounts on its machine. The user interacts with the transaction manager, which in turn uses the data managers and a stable storage object to implement transactions. The transaction manager acts as the coordinator of the system; it decides if and when a transaction is to be committed and coordinates the two-phase commit protocol [Gra79] that is used to ensure that all data managers involved agree on the outcome of the transaction. Associated with the transaction manager is a stable storage object, which it uses to log the progress of transactions in the system. The data managers export operations that are used by the transaction manager to read and write user accounts, and to implement the two-phase commit protocol. The stable storage associated with each data manager is used to store the actual data corresponding to user accounts, and to maintain key values that can be used to restore the state of the data manager should a failure occur. The lock managers are used to control concurrent access.

To increase the overall dependability of the system, the constituent FS atomic objects would typically be constructed using fault-tolerance techniques to increase their failure resilience. For example, the transaction and data managers might use recovery protocols to ensure that data in the system is restored to a consistent state following failure. Similarly, stable storage might be replicated to increase its failure resilience. The failure notification aspect of FS atomic objects is used to allow objects to react to the failures of objects upon which they depend. If such a failure cannot be tolerated, it may, in turn, cause subsequent failures to be propagated up the dependency graph. At the top level, this would be viewed by the user as the catastrophic failure of the transaction manager and hence, the system. Such a situation might occur, for example, should the redundancy being used to implement stable storage be exhausted by an untimely series of failures. In Section 4, we illustrate how the data manager and stable storage FS atomic objects from this example might be implemented using FT-SR.

Fail-stop atomic objects and the associated techniques for increasing failure resilience form, in our view, a "lowest common denominator" that can be conveniently used to realize the seemingly disparate programming paradigms proposed for fault-tolerant programming. For example, consider the object/action model. As its name implies, there are two types of components, objects and actions. Objects are passive entities that encapsulate a state and export certain operations to modify that state; typically, this state involves long-lived data that is assumed to be stored on stable storage to survive failures. Actions are active entities similar to threads that invoke operations on objects to carry out some task. The objects comprising an application can potentially be located on multiple machines in a network, which implies that actions may logically cross machine boundaries during their execution. An action is both unitary and serializable, which guarantees the atomicity of its execution with respect to both failures and the concurrent execution of other actions. These properties have also been called totality and serializability [Wei89], and recoverability and indivisibility [Lis85]. In the database literature, atomic actions are known as transactions [BHG87].

A system built using the object/action paradigm may be implemented using FS atomic objects. Objects in the system correspond to FS atomic objects. An action corresponds to an abstract thread realized by the combination of concrete threads in the FS atomic objects. This abstract thread may span multiple FS atomic objects as a result of invocations made by concrete threads that are serviced by concrete threads in other objects. Standard locking and commit protocols are used to ensure the unitary and serializable nature of these actions across multiple objects. Viewed as a whole, this system appears to the user as one FS atomic object exporting the set of operations required by the application.

As a second example, consider the replicated state machine paradigm. In this paradigm, a system is implemented as a collection of interacting state machines. Each such machine consists of *state variables*, which encode its state, and *commands*, which transform its state. Each command is implemented by a deterministic program that modifies the state variables and/or produces some output. Command executions are atomic and any outputs produced are determined solely by the sequence of commands processed by the state machine. A fault-tolerant version of a state machine can be implemented by replicating that state machine and running each replica on a different processor in a distributed system. All available replicas must receive all commands sent to the replicated state machine (the *agreement* property) and must process them in the same sequence (the *order* property).

State machines map directly to FS atomic objects. Commands to the state machine correspond to invocations on an FS atomic object, with locking techniques being used to ensure that the operation executions are atomic. A replicated state machine can be implemented by replicating FS atomic objects and ensuring that all commands to the state machine result in invocations on all replicas in a consistent order. Each ensemble of replicated FS atomic objects forms a higher-level FS atomic object representing the fault-tolerant version of a given state machine. The entire collection of such FS atomic objects can then be viewed as a single FS atomic object that implements the entire system.

The system shown in Figure 1 is an example of a system in which FS atomic objects are used to implement different programming paradigms in different parts of the system. Specifically, the transaction and data managers are built using the restartable action paradigm, while the stable storage objects are built using the replicated state machine approach. The user of the banking system sees the system as one implementing the atomic object/action paradigm and interacts with it accordingly.

3 FT-SR Language Description

The goal of FT-SR is to support the building of systems based on the FS atomic object model, and thus, by implication, the building of systems using any of the existing programming paradigms. Given the need for flexibility, we do not provide these objects directly in the language, but rather include features that allow them to be easily implemented. To this end, the language has provisions for encapsulation based on SR resources, resource replication, recovery protocols, synchronous failure notification when performing interprocess communication, and a mechanism for asynchronous failure notification based on a previous scheme for SR [SCP91]. Since our extensions are based on existing SR mechanisms, a short overview of the language is provided in Appendix A; for further details, see [AOC+88, AO93].

3.1 Simple FS Atomic Objects

Realizing much of the functionality of a simple FS atomic object—i.e., one not composed of other objects or using any other fault-tolerance techniques—in SR is straightforward since a resource instance is essentially an object in its own right. For example, it has the appropriate encapsulation properties, and is populated by a varying number of processes that can function as threads in the FS atomic object model. SR operations are also very similar to the operations defined by the model; they are implemented by processes and can be exported for invocation by processes in other resource instances. Moreover, the execution semantics of SR operations are already close to those desired for FS atomic objects; the only additional property required is atomicity of operation execution in the absence of catastrophic failures, which occur for simple objects when the resource instance is destroyed due to failure or explicit termination. Ensuring atomicity therefore reduces to ensuring serializability, which can easily be programmed in SR by, for example, implementing each exported operation as a separate alternative in an input statement repeatedly executed by a single process. Standard locking-based solutions that allow more concurrency are also easy to implement in SR.

Figure 2 shows the outline of a simple FS atomic object that implements atomicity by use of an SR input statement. The object is a lock manager that controls access to a shared data structure implemented by some other resource. It exports two operations: a `get_lock` operation that is invoked by clients wishing to access the shared data structure and a `rel_lock` operation that clients invoke when they are done. If a client invokes the `get_lock` operation and the lock is available, a `lock_id` is returned. If the lock is unavailable, the client is blocked at the first guard of the input statement. The `get_lock` operation takes as its argument the capability of the invoking client. This capability is used as a means of identifying the client.

The one aspect of simple FS atomic objects that SR does not support directly—and hence the focus of our extensions in this area—is generation of a failure notification. As mentioned earlier, for simple objects this occurs when the processor executing the resource instance fails, or when the resource instance or its virtual machine is explicitly destroyed from within the program. In Section 5, we discuss how this failure is detected by the language runtime system, so here we concentrate on describing the mechanisms that are provided to field this notification in other resource instances. These facilities allow an abstract object to react to the failure of other objects on which it depends.

FT-SR provides the programmer with two different kinds of failure notification and consequently, two different ways of fielding a notification. The first is synchronous with respect to a call; it is fielded by an optional *backup operation* specified in the calling statement. The second kind of notification is asynchronous; the programmer specifies a resource to be monitored and an operation to be invoked should the monitored resource fail. To understand the need for these two kinds of failure notification, consider what might happen if the lock manager shown in Figure 2 or any of its clients fail. If the lock

```
resource lock_manager
    op get_lock(cap client) returns int
    op rel_lock(int)
body lock_manager
    var ...variable declarations...

    process lock_server
        do true ->
            in get_lock(client_cap) and lock_available() ->
                ...mark lock_id as being held by client_cap...
                return lock_id

            [] rel_lock(client_cap, lock_id) ->
                ...release lock...
                return
            ni
        od
    end lock_server
end lock_manager
```

Figure 2: Simple FS atomic object

manager fails, all clients that are blocked on its input statement will remain blocked forever. Clients can use the FT-SR synchronous failure notification facility to unblock themselves from the call and take some recovery action in the event of such a failure. Figure 3 shows the outline of a client structured in this way. The statement of interest is where the client makes a call invocation using a capability to the lock manager's `get_lock` operation `lock_mgr_cap.get_lock`. Bracketed with this capability is the capability to a backup operation `mgr_failed`. This backup operation is invoked should the call to `lock_mgr_cap.get_lock` fail, where the call is defined to have failed if the lock manager fails before it can reply to the call. In this example, the backup operation `mgr_failed` is implemented locally by the client, which we expect will be the most common usage; in general, however, the backup operation can be implemented by any resource. Note that the backup operation is called with the same arguments as the original operation and, hence, must be type compatible with the original operation. Backup operations can only be specified with call invocations; send invocations are non-blocking and no guarantees can be made about the success or failure of such an invocation if the resource implementing the operation fails. Both call and send invocations are guaranteed to succeed in the absence of failures. Execution is blocked if a call fails and there is no associated backup operation.

Consider now the inverse situation where a client of the lock manager fails. If the client fails while it holds a lock, all other clients will be prevented from accessing the shared data structure. The server can use the FT-SR asynchronous failure notification facility to detect such a failure and release the lock, as shown in Figure 4. This figure is identical to Figure 2 except for the **`monitor`** statement in the `get_lock` operation and the **`monitorend`** statement in the `rel_lock` operation. The server uses the **`monitor`** statement to enable monitoring of the client instance specified by the resource capability `client_cap`. If the client is down when the statement is executed or should it subsequently fail, the operation `rel_lock` will be implicitly invoked by the language runtime system with the `client_cap` and `lock_id` as arguments. Arguments to the operation specified in the **`monitor`** statement are evaluated at the time the **`monitor`** statement is executed and not when the failure occurs. Monitoring is terminated by the **`monitorend`** statement, which also takes a resource capability as its argument (as shown in Figure 4) or by another **`monitor`** statement that specifies the same resource. The ability to request asynchronous notification has proven to be convenient in a variety

```
resource client
    op ...
    op ...
body client()
    var lock_id:  int
    op mgr_failed(cap client) returns int
    ⋮
    lock_id := call {lock_mgr_cap.get_lock, mgr_failed} (myresource())
    ⋮

    proc mgr_failed(client_cap) returns lock_err
        return LOCK_ERR
    end mgr_failed
end client
```

Figure 3: Outline of Lock Manager client

```
resource lock_manager
    op get_lock(cap client) returns int
    op rel_lock(int)
body lock_manager
    var ...variable declarations...

    process lock_server
        do true ->
            in get_lock(client_cap) and lock_available() ->
                ...mark lock_id as being held by client_cap...
                monitor client_cap send rel_lock(client_cap, lock_id)
                return lock_id

            [] rel_lock(client_cap, lock_id) ->
                ...release lock if held by client_cap...
                monitorend client_cap
                return
            ni
        od
    end lock_server
end lock_manager
```

Figure 4: Lock Manager with client monitoring

of contexts [CGR88, SCP91, BMZ92] and is in keeping with the inherently asynchronous nature of failures themselves.

3.2 Higher-Level FS Atomic Objects

FT-SR provides mechanisms for supporting the construction of more fault-tolerant, higher-level FS atomic objects using replication, and for increasing the resilience of objects to failures using recovery techniques. The replication facilities allow multiple copies of an FT-SR resource to be created, with the language and runtime providing the illusion that the collection is a single resource instance exporting the same set of operations. The SR create statement has been generalized to allow for the creation of such replicated resources, which we call a *resource group*. For example, the statement

```
lock_mgr_cap := create (i := 1 to N) lock_manager() on vm_caps[i]
```

creates a resource group with N identical instances of the resource `lock_manager` on the virtual machines specified by the array of virtual machine capabilities `vm_caps`. Both the quantifier `(i := 1 to N)` and **`on`** clauses are optional. If they are omitted the statement reverts to the semantics of the normal SR statement, which creates one instance of the named resource on the current virtual machine.

The value returned from executing the **`create`** statement is a resource capability that provides access to the operations implemented by the new resource(s). If a single resource instance is created, the capability allows the holder to invoke any of the exported operations in that instance as provided for in normal SR. If, on the other hand, multiple identical instances are created, the capability is a *resource group capability* that allows multicast invocation of any of the group's exported operations. In other words, using this capability in a **`call`** or a **`send`** statement causes the invocation to be multicast to each of the individual resource instances that make up the group. All such invocations are guaranteed to be delivered to the runtime of each instance in a consistent total order, although the program may vary this if desired. This means, for example, that if two operations implemented by alternatives of an input statement are enabled simultaneously, the order in which they will be executed is consistent across all functioning replicas unless a scheduling expression by the programmer overrides this explicitly. Moreover, the multicast is also done atomically, so that either all replicas receive the invocation or none do. This property is guaranteed by the runtime given no greater than *max_sf* simultaneous failures, where *max_sf* is a parameter set by the user at compile time. The combination of the atomicity and consistent ordering properties means that an invocation using a resource group capability is equivalent to an *atomic broadcast* [CAS85, MSMA90]. The results of a multicast call invocation are collected by the runtime system, with only a single result being returned to the caller; since FT-SR assumes processors with fail-silent semantics, returning the first result is sufficient in this case.

In addition to this facility for dealing with invocations coming into a resource group, provisions are also made for coordinating outgoing invocations generated within the group. There are two kinds of invocations that can be generated by a group member. In some cases, a group member may wish to communicate with a resource instance as an individual even though it happens to be in a group. For example, this would be the situation if each replica has its own set of private resources with which it communicates. At other times, the group members might want to cooperate to generate a single outgoing invocation on behalf of the entire group. To distinguish between these two kinds of communication, FT-SR allows a capability variable to be declared as being of type **`private cap`**. Invocations made using a private capability variable are considered private communication of the group member and not co-ordinated with other invocations from group members. Invocations using regular capability variables are, however, considered to be invocations from the entire group, so exactly one invocation is generated in this case. The invocation is actually transmitted when one of the group members reaches the statement, with later instances of the same invocation being suppressed by the language runtime system. This invocation could, in fact, be a multicast-type invocation as described above if the operation being invoked is within another resource group (i.e., if the capability used in the statement is a resource group capability). It should be noted that a private capability variable can be assigned to a regular capability of the same type and vice versa; whether an invocation is private or not is determined solely by the type of the variable used in making the invocation.

A resource group can also be configured to work according to a primary-backup scheme [AD76]. In this scenario, invocations to the group are delivered only to a replica designated as the primary by the language runtime, with the other replicas being passive. This type of configuration is achieved by placing the op restrictor {**primary**} on the declaration of operations in the group members that are to be invoked only if the replica is the primary.

FT-SR also provides the programmer with the ability to restart a failed resource instance on a functioning virtual machine. The recovery code to be executed upon restart is denoted by placing it between the keywords **recovery** and **end** in the resource text. This syntax is analogous to the provisions for initialization and finalization code in the standard version of SR. A resource instance may be restarted either explicitly or implicitly. Explicitly, it is done by the following statement:

```
restart lock_mgr_cap() on vm_cap
```

This restarts the resource indicated by the capability `lock_mgr_cap` and executes any recovery code that may be specified by the programmer. To restart an entire resource group,

```
restart (i:=1 to N) lock_mgr_cap() on vm_caps[i]
```

is used. The size of the reconstituted group can be different from the original. In both cases, it is important to note that the restarted resource instance is, in fact, a *re-creation* of the failed instance and not a new instance. This means, for example, that other resource instances can invoke its operations using any capability values obtained prior to the failure.

Implicit restart is indicated by the presence of the keyword **persistent** in the resource declaration and the inclusion of more virtual machines in the `vm_caps` array specified in the original create statement than the number of replicas actually created. Then, should a virtual machine executing one of the instances of the resource group fail, the system will select one of these *backup virtual machines* and recreate the failed instance automatically. The arguments supplied during the recreation are the same as those used for the original creation. This facility is designed to allow a resource group to automatically regain its original level of redundancy following a failure.

Another issue concerning restart is determining when the runtime of the recovering resource instance begins accepting invocations from other instances. In general, the resource is in an indeterminate state while performing recovery, so we choose to begin accepting messages only after the recovery code has completed. The one exception to this is if the recovering instance itself initiates an invocation during recovery; in this case, invocations are accepted starting at the point that particular invocation terminates. This is to facilitate a system organization in which the recovering instance retrieves state variables from other resources during recovery.

Finally, we note that the failure notification facilities described in the previous section work with resource groups as one would expect. For such higher-level FS atomic objects, a catastrophic failure occurs when all of the replicas have been destroyed by failure or explicit termination request(s), and there is no system guarantee of recreation. Thus, if a resource is not persistent, a notification is generated once all replicas have been destroyed, while for a persistent resource, a notification is generated once all replicas have been destroyed and the list of backup virtual machines exhausted. In either case, the way in which the notification is fielded is specified using backup operations or the **monitor** statement in the same way as before.

4 Programming with FT-SR

In this section, we present an example program that illustrates how FT-SR can be used to construct a simple system that uses multiple fault-tolerance paradigms. The example consists of the data manager and stable storage objects from the banking system described in Section 2. As outlined there, the data manager implements a collection of operations that provide transactional access to data items located on a stable storage. The organization of the manager itself is based on the restartable action paradigm, with key items in the internal state being saved on stable storage for later recovery in the event of failure. The state machine approach is used to build stable storage. A prototype version of this banking system has been implemented and is currently being tested.

The data manager controls concurrency and provides atomic access to data items on stable storage. For simplicity, we assume that all data items are of the same type and are referred to by a logical address. Stable storage is read by invoking its `read` operation, which takes as arguments the address of the block to be read, the number of bytes to be read, and a buffer in which the values read are to be returned. Data is written to stable storage by invoking an analogous `write` operation, which takes as arguments the starting address of the block being written, the number of bytes in the block, and a buffer containing the values to be written.

Figures 5 shows the specification and an outline of the body of such a data manager. As can be seen in its specification, the data manager imports stable storage and lock manager resources, and exports six operations: `startTransaction`, `read`, `write`, `prepareToCommit`, `commit`, and `abort`. The operation `startTransaction` is invoked by the transaction manager to access data held by the data manager; its arguments are a transaction identifier `tid` and a list of addresses of the data items used during the transaction. `read` and `write` are used to access and modify objects. The two operations `prepareToCommit` and `commit` are invoked in succession upon completion to, first, commit any modifications made to the data items by the transaction, and, second, terminate the transaction. `abort` is used to abandon any modifications and terminate the transaction; it can be invoked at any time up to the time `commit` is first invoked. All of these operations exported by the data manager are implemented as **`procs`**; thus, invocations result in the creation of a thread that executes concurrently with other threads. Finally, the data manager contains initial and recovery code, as well as a failure handler **`proc`** that deals with the failure of the `lockManager` and `stableStore` resources.

To implement the atomic update of the data items, the data manager uses the standard technique of maintaining two versions of each data item on stable storage together with an indicator of which is current [BHG87]. To simplify our implementation, we maintain this indicator and the two versions in contiguous stable storage locations, with the indicator being an offset and the address of the indicator used as the logical address of the item. Thus, the actual address of the current copy of the item is calculated by taking the address of the item and adding to it the indicator offset.

The data manager keeps track of all in-progress transactions in a *status table*. This table contains for each active transaction the transaction identifier (`tid`), the status (`transStatus`), the stable storage addresses of the data items being accessed by the transaction (`dataAddrs`), the value of the indicator offset of each item (`currentPointers`), a pointer to an array in volatile memory containing a copy of the data items (`memCopy`), and the number of data items being used in the transaction (`numItems`).

```
resource dataManager
    imports globalDefs, lockManager, stableStore
    op startTransaction(tid: int; dataAddrs: addrList; numDataItems: int)
    op read(tid: int; dataAddrs: addrList; data: dataList; numDataItems: int)
    op write(tid: int; dataAddrs: addressList; data: dataList; numDataItems: int)
    op prepareToCommit(tid: int), commit(tid: int), abort(tid: int)
body dataManager(dmId: int; lmcap: cap lockManager; ss: cap stableStore)
    type transInfoRec = rec(tid: int;
                transStatus: int;
                dataAddrs: addressList;
                currentPointers: intArray;
                memCopy: ptr dataArray;
                numItems: int)
    var statusTable[1:MAX_TRANS]: transInfoRec; statusTableMutex: semaphore

    initial
        # initialize statusTable
            ...
        monitor(ss)send failHandler()
        monitor(lmcap)send failHandler()
    end initial

    ...code for startTransaction, prepareToCommit, commit, abort, read/write...

    proc failHandler()
        destroy myresource()
    end failHandler

    recovery
        ss.read(statusTable, sizeof(statusTable), statusTable);
        transManager.dmUp(dmId);
    end recovery
end dataManager
```

Figure 5: Outline of `dataManager` resource

This table can be accessed concurrently by threads executing the **procs** in the body of the data manager, so the semaphore `statusTableMutex` is used to achieve mutual exclusion. New entries in this table also get saved on stable storage for recovery purposes. Reads and writes during execution of the transaction are actually performed by the data manager on versions of the items that it has cached in its local (volatile) storage.

The data manager depends on the stable storage and lock manager resources to implement its operations correctly. As a result, it needs to be informed when they fail catastrophically. The data manager does this by establishing an asynchronous failure handler `failHandler` for both of these events in the initial code using the **monitor** statement. When invoked, `failHandler` terminates the data manager resource, thereby causing the failure to be propagated to the transaction manager.

The failure of the data manager itself is handled by recovery code that retrieves the current contents of the status table from stable storage. It is the responsibility of the transaction manager to deal with transactions that were in progress at the time of the failure; those for which `commit` had not yet been invoked are aborted, while `commit` is reissued for the others. To handle this, the recovery code sends a message to the transaction manager notifying it of the recovery.

The **procs** implementing the other data manager operations do not use any of the FT-SR primitives specifically designed for fault-tolerant programming and are therefore not shown here. For completeness, they can be found in Appendix B.

We now turn to implementing stable storage. One way of realizing this abstraction is by using

```
persistent resource stableStore
    import globalDefs
    op read(address:  int; numBytes:  int; buffer:  charArray)
    op write(address:  int; numBytes:  int; buffer:  charArray)
    op sendState(sscap:  cap stableStore)
    op recvState(objectStore:  objList)
body stableStore
    var store[MEMSIZE]: char

    process ss
        do true ->
            in read(address, numBytes, buffer) ->
                buffer[1:numBytes] := store[address:address+numBytes-1]
            □ write(address, numBytes, buffer) ->
                store[address, address+numBytes-1] := buffer[1:numBytes]
            □ sendState(rescap) -> send rescap.recvState(store)
            ni
        od
    end ss

    recovery
        send mygroup().sendState(myresource())
        receive recvState(store); send ss
    end recovery
end stableStore
```

Figure 6: `stableStore` resource

the state machine approach, that is, by creating a storage resource and replicating it to increase failure resilience. Figure 6 shows such a resource; for simplicity, we assume that storage is managed as an array of bytes.

Replica failures are dealt with by restarting the resource on another machine; this is done automatically since `stableStore` is declared to be a persistent resource. The recovery code that gets executed in this scenario starts by requesting the current state of the store from the other group members. All replicas respond to this request by sending a copy of their storage state; the first response is received, while the other responses remain queued at the `recvState` operation until the replica is either destroyed or fails. The newly restarted replica begins processing queued messages when it is finished with recovery. Since messages are queued from the point that its `sendState` message was sent to the group, the replica can apply these subsequent messages to the state it receives to reestablish consistency with the state of the other replicas.

Stable storage could also be implemented as a primary-backup group by adding a {**primary**} restriction to the `read` and `write` operations. The process `ss` would then send the updated state to the rest of the group at the end of each operation by invoking a `recvState` operation on the group. This operation would be implemented by extending the input statement in `ss` to include this operation as an additional alternative.

The main resource that starts up the entire system is shown in Figure 7. Resource `main` creates a virtual machine on each of the three physical machines available in the system. Three stable storage objects are then created, where each such object has two replicas and uses the virtual machine on "host3" as a backup machine. The two data managers are then created followed by the transaction manager. Notice how the system is created "bottom up," with the objects at the bottom of the dependency graph being created before the objects on which they depend. This way, each object can be given capabilities to the objects on which it depends upon creation.

```
resource main
    imports transManager, dataManager, stableStore, lockManager
body main
    var virtMachines[3] :  cap vm # array of virtual machine capabilities
    dataSS[2], tmSS: cap stableStore # capabilities to stable stores
    lm:  cap lockManager; dm[2]:  cap dataManager # capabilities to lock and data managers

    virtMachines[1] := create vm() on ''host1''
    virtMachines[2] := create vm() on ''host2''
    virtMachines[3] := create vm() on ''host3'' # backup machine

    # create stable storage for use by the data managers and the transaction manager
    dataSS[1] := create (i := 1 to 2) stableStore() on virtMachines
    dataSS[2] := create(i := 1 to 2) stableStore() on virtMachines
    tmSS := create (i := 1 to 2) stableStore() on virtMachines

    # create lock manager, data managers, and transaction manager
    lm := create lockManager() on virtMachines[2]
    fa i := 1 to 2 ->
        dm[i] = create dataManager(i, lm, dataSS[i]) on virtMachines[i]
    af
    tm = create transManager(dm[1], dm[2], tmSS) on virtMachines[1]
end main
```

Figure 7: System startup in resource `main`

5 Implementation

Overview. The implementation of FT-SR consists of two major components: a compiler and a runtime system. Both the compiler and the runtime system are written in C and borrow heavily from the existing implementation of SR. In fact, the FT-SR compiler is almost identical to the SR compiler, which is to be expected since FT-SR is syntactically close to SR. The compiler is based on lex and yacc, and consists of about 16,000 lines of code. It generates C code, which is in turn compiled by a C compiler and linked with the FT-SR runtime system.

The FT-SR runtime system provides primitives for creating, destroying and monitoring resources and resource groups, handling failures, restarting failed resources, invoking and servicing operations, and a variety of other miscellaneous functions. It consists of 9600 lines of code and is implemented using version 3.1 of the *x*-kernel, a stand-alone operating system kernel that runs on Sun 3s [HP91]. The major advantage of such a bare machine implementation is that it gives us the ability to use FT-SR to build realistic fault-tolerant systems and experiment with these systems by actually crashing and restarting processors. This is in contrast to experimental systems built, for example, on top of Unix. In addition, the *x*-kernel provides a flexible infrastructure for composing communication protocols; this has proven to be extremely useful in building the many protocols that went into the FT-SR runtime system.

Figure 8 shows the organization of the FT-SR runtime system on a single processor. As shown in the figure, each FT-SR virtual machine exists in a separate *x*-kernel user address space. In addition to the user program, a virtual machine contains those parts of the runtime system that create and destroy resources, route invocations to operations on resources, and manage intra-virtual machine communication. This user resident part accounts for about 85 percent of the runtime system. The remaining 15 percent resides inside the kernel and is responsible for the creation and destruction of virtual machines and inter-virtual machine communication. Figure 8 also shows some of the important modules in both the kernel and user resident parts of the runtime system, and the communication

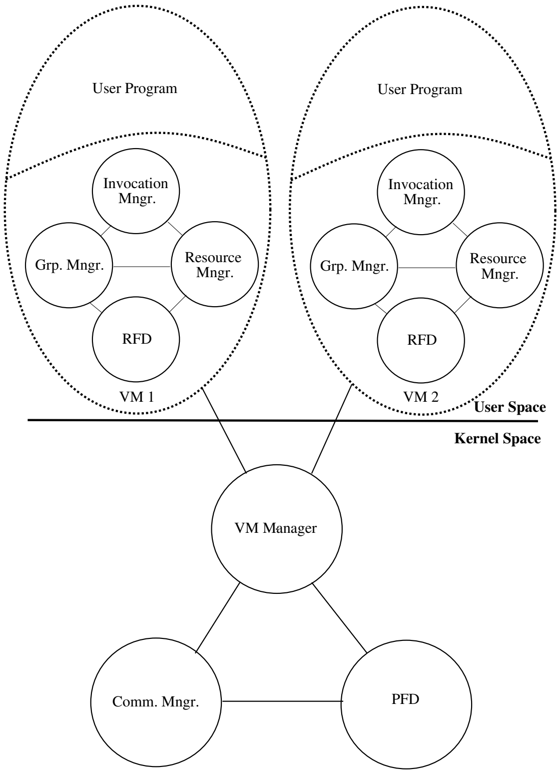

Figure 8: Organization of FT-SR runtime system

paths between them. The kernel resident modules shown are the Communication Manager, the Virtual Machine (VM) Manager, and the Processor Failure Detector (PFD). The communication manager consists of multiple communication protocols that provide point-to-point and broadcast communication services between processors. The VM Manager is responsible for creating and destroying virtual machines, and for providing communication services between virtual machines. The PFD is a failure detector protocol; it monitors processors and notifies the VM manager when a failure occurs. The user space resident modules shown in Figure 8 are the Resource Manager, the Group Manager, the Invocation Manager, and the Resource Failure Detector (RFD). The Resource Manager is responsible for the creation, destruction and restart of failed resources. The Group Manager is responsible for the creation, destruction and restart of groups, restart of failed group members, and all communication to and from groups. The RFD detects the failure of resources and group members.

The FT-SR runtime system, while similar in structure to that of standard SR, differs significantly in terms of how its different constituent modules implement their functionality. The reasons for this difference are threefold. First, unlike SR, which implements its own threads package within a Unix process, FT-SR uses threads provided by the *x*-kernel. Therefore, while SR has complete control over its threads and can schedule them when it deems safe, the FT-SR runtime has no control over when a thread is preempted and another thread scheduled. This gives rise to numerous concurrency problems that had to be dealt within the FT-SR implementation. Second, the need to deal with failures required large modifications to the SR runtime system, since it uses centralized control in places and is written under the assumption that all parts of the distributed program will always be available. The FT-SR runtime system cannot make such an assumption, and this need to anticipate and deal with failures affected the design of almost every runtime system module. Finally, in addition to the standard language features, the FT-SR runtime system had to, of course, provide support for the fault-tolerance extensions. The implementation of some of these extensions required new modules within the runtime system and at times the redesign of other modules.

From the programmer's viewpoint, there are two major aspects in which FT-SR differs from SR: the support for using resources as FS atomic objects, and the support for various fault-tolerance techniques. Accordingly, we focus in the remainder of this section on describing those parts of the FT-SR language runtime system that provide this support. It is worth keeping in mind that, since FT-SR is designed to support construction of fault-tolerant systems, every effort has been made to keep the implementation as efficient as possible. For example, whenever possible, the implementation takes advantage of the fact that the processors can only suffer from fail-silent failures and that the maximum number of simultaneous failures *max_sf* is known *a priori*.

Failure Detection and Notification. Failure detection and notification is the single most important difference between FS atomic objects and SR resources. Failure detection is initiated in one of two ways: when a resource explicitly asks to be notified of a failure using the **`monitor`** statement, or when the communication module of the runtime system cannot complete an invocation and suspects a failure. Depending on how the failure detection was initiated, the runtime system either notifies the user program of the failure by generating an implicit invocation of the operation specified by the **`monitor`** statement, or, if a backup operation has been supplied with an invocation, the invocation is forwarded to the backup operation.

Failure detection in FT-SR is done at two levels: at the virtual machine level by the VM Manager and at the resource level by the RFD. The PFD at each processor monitors the other processors in the system and notifies the local VM manager of any failures. The VM manager then maps these processor failures to failures of virtual machines and notifies the RFD of these failures. The RFD in turn maps virtual machine failures to resource failures and passes this information on to any of the other runtime system modules that might have asked to be notified of the failure. To detect the termination of a resource that is explicitly destroyed, the RFD sends a message to its peer on the appropriate virtual machine asking to be notified when the resource is destroyed. Similarly, a VM Manager can ask another VM Manager to send a failure notification when a virtual machine is explicitly destroyed.

Replication and Recovery. FT-SR provides two mechanisms for increasing failure resilience: replication and recovery. For replication, the most interesting aspect of the implementation is managing group communication, since messages sent to a resource group as a result of invocations have to be multicast and delivered to all replicas in a consistent total order. The technique we use is similar to [CM84, KTHB89, GMS89], where one of the replicas is a *primary* through which all messages are funneled. Another *max_sf* replicas are designated as *primary-group members*, with the remaining being considered *ordinary members*. Upon receiving a message, the primary adds a sequence number and multicasts it to all replicas of the group. Only the replicas that belong to the primary-group acknowledge receipt of the message. As soon as the primary gets these *max_sf* acknowledgements, it sends an acknowledgement to the original sender of the message; this action is appropriate since the receipt of *max_sf* acknowledgements guarantees that at least one replica will have the message even with *max_sf* failures. The primary is also involved when messages are sent by the group as a whole, that is, when group members use a capability that is not a private capability when making a invocation. The runtime system suppresses such an invocation from all group members except the primary. When the primary receives an acknowledgement that its invocation has been received, it multicasts that acknowledgement to the other group members.

The Group Managers at each site are responsible for determining the primary and the members of the primary-group set. Specifically, they maintain a list of all the group members and keep track of which member is the primary, which belong the primary-group set, and which are ordinary members. This list is ordered consistently at all sites based on the order in which the replicas were specified in the group create statement. The first replica of the list is then designated the primary and the next *max_sf* as members of the primary-group; the remaining replicas are ordinary members. The consistent ordering of replicas ensures that all Group Managers will independently pick the same primary and assign the same set of replicas to the primary-group set.

The Group Managers are also responsible for dealing with the failure of group members. The action taken when a failure occurs varies, depending on whether the failed member was the primary, a primary-group member, or an ordinary member. If the primary fails, the first member of the primary-group is designated as the new primary. The designation of a new primary or the failure of a primary-group member will cause the size of the primary-group to fall below *max_sf*. When this happens, ordinary members are added to the primary-group to bring it up to *max_sf* members. No special action is needed when an ordinary member fails. If the resource from which the failed member was created is declared as being persistent and backup virtual machines were specified in the create statement, failed replicas are restarted on these backups. Restarted replicas join the group as ordinary members.

Supporting recovery involves: (1) restarting the resource instance, either as a result of an explicit request or due to its declaration as persistent, (2) ensuring that the recovery code is executed, and (3) correctly starting the delivery of new invocations. Actually implementing (1) and (2) are fairly easy since the requirements for restarting a resource instance are very similar to creating one initially. For a persistent resource, this is preceded by the selection of a backup virtual machine from the list supplied during initial creation to act as the new host. The policy used to ensure (3) has already been described in Section 3.2.

6 Discussion

6.1 FT-SR and Fault-Tolerance Abstractions

The various programming paradigms that have been developed for fault-tolerant distributed systems provide the programmer with structuring techniques and abstractions for the problem being solved. The relationship between the abstractions used for a particular paradigm and the mechanisms provided by FT-SR can be illustrated by arranging them in a hierarchy based on the dependency relationship [MS92]. Figure 9 shows such a hierarchy for the object/action model. In this figure, the circles represent abstractions, the rectangles represent mechanisms provided by FT-SR, and the labeled boxes at the bottom represent the portions of the FT-SR runtime system that implement these language mechanisms. At the top of the hierarchy are objects and atomic actions. These depend directly on restartable actions, which are needed to implement protocols like the two-phase commit protocol in which failed processes must recover to successfully complete. These restartable actions, in turn, depend on two other abstractions: *idempotent actions* and stable storage. Idempotent actions are actions that can be restarted if interrupted by failure without the need to restore the initial state; for example, writing to stable storage can be implemented in this way using an *intentions list* [Lam81]. Stable storage itself is, of course, used to maintain the intermediate states of atomic actions and other key values; in most cases, stable storage is realized in hardware (e.g., a disk), but, as shown in Section 4, it is possible to build such an abstraction in software using replicated state machines or primary/backup schemes.

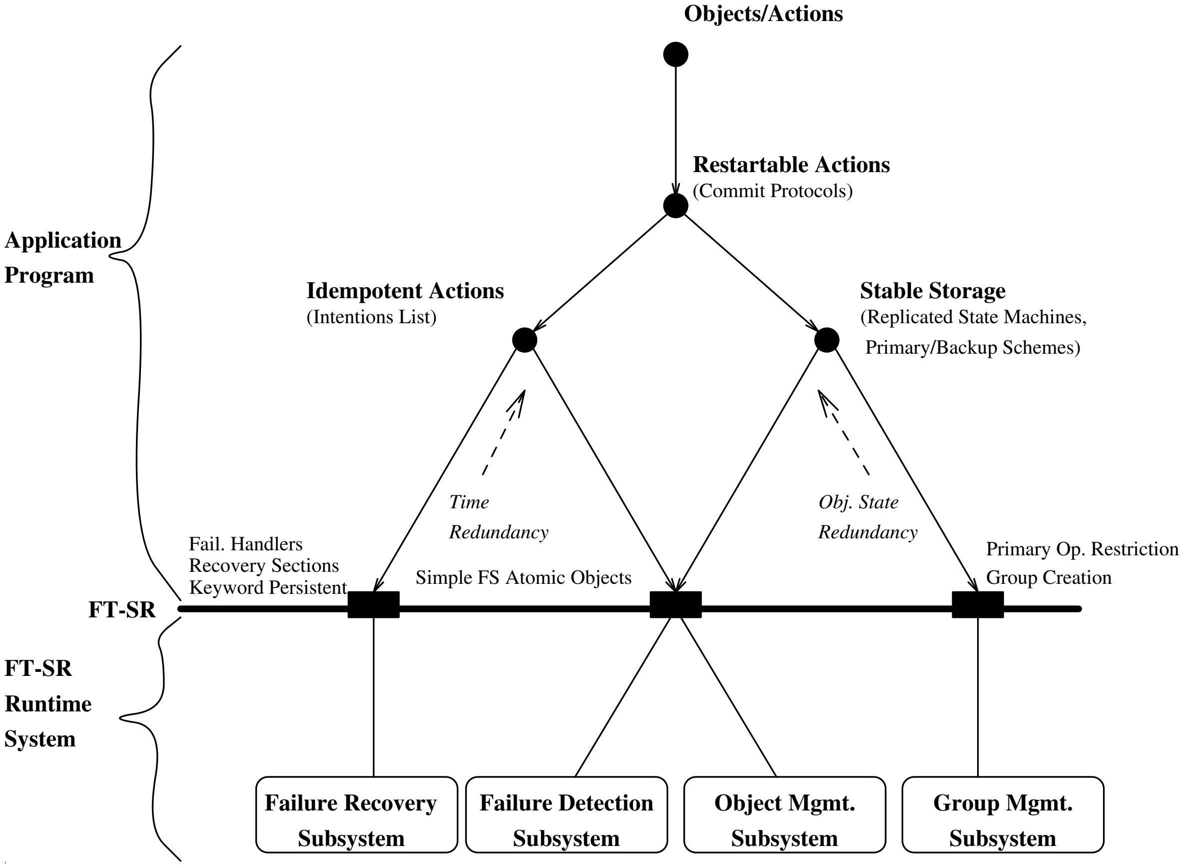

Figure 9: Abstraction hierarchy for object/action model

Finally, the mechanisms and runtime modules in FT-SR are at the base of the hierarchy,

Similar hierarchies can be defined for programming paradigms such as restartable actions and the state machine approach, and in each, FT-SR is the foundation. As a result, we argue that FT-SR provides the fundamental abstractions that underlie fault-tolerant programming. In particular, FT-SR provides the abstraction of a simple FS atomic object and two categories of language mechanisms: those that deal with failure recovery and those that deal with group management. A simple FS atomic object may be combined with these language mechanisms in a variety of ways to realize the abstractions needed to implement the different programming paradigms. For example, as illustrated in our banking example, an object with a recovery section can be used to implement a restartable action. Similarly, the FT-SR **`create`** statement can be used to replicate objects to construct stable storage. In addition, FT-SR gives the programmer the ability to create groups where the degree of replication is automatically maintained and the ability to restart entire groups after a catastrophic failure.

We also feel that FT-SR provides an appropriate level of abstraction for a systems programming language. Its mechanisms are primitive enough to give the programmer the ability to build all other abstractions, yet powerful enough to be able to do so with relative ease. Such flexibility allows FT-SR to be used for a variety of different applications and system architectures. The primitive nature of the mechanisms also allows them to be efficiently implemented, an important consideration for a systems programming language.

6.2 Related Work

Other programming languages that have been proposed for fault-tolerant programming differ from FT-SR in that they provide support for only a single paradigm, making it difficult to construct applications using other approaches. For example, languages like Argus [Lis85], Avalon [HW87], Plits [EFH82], TABS [SDD+85] and Hops [Mad86] support the object/action model, languages like Fault-Tolerant Concurrent C [CGR88] support the replicated state machine approach, and numerous operating systems like UNICOS [KK89] support the restartable action paradigm. To illustrate how the different programming paradigms are typically supported, we describe a representative example from each category: Fault-Tolerant Concurrent C, UNICOS and Argus.

Fault-Tolerant Concurrent C. Fault-Tolerant Concurrent C (FTCC) is a collection of extensions to Concurrent C [GR89] that allows the programmer to build replicated state machines using multiple processes. This is done by extending the Concurrent C process creation statement to create multiple copies of a process on one or more processors. The entire replicated process ensemble is identified by a single process identifier and calls using this identifier are delivered by the FTCC runtime system to all members of the ensemble. A distributed consensus protocol in the runtime system ensures that all replicas process messages in the same total order. While FTCC does not provide the programmer with any mechanism to restart failed replicas on functioning processors, it does provide the ability to detect process failures, where a process is defined to have failed if all of its replicas are lost due to failures or are destroyed by the program. The programmer can ask to be synchronously or asynchronously notified of such failures. Synchronous failure notification is provided by means of a *fault-expression* associated with the call statement that is executed if the call fails. Asynchronous failure notification is provided by the built-in function `c_request_death_notice`, which takes two arguments: the identifier of a process to be monitored and a function to be called by the runtime system when the process fails. This monitoring can be terminated using the built-in function `c_cancel_death_notice`, which takes a process identifier as its argument.

FTCC's support for the replicated state machine approach is suitable for building a large class of fault-tolerant applications. However, there is an equally large class for which some other paradigm is most convenient, and for these, FTCC provides no explicit support. FT-SR, on the other hand, provides an integrated collection of mechanisms that supports a variety of paradigms equally well.

UNICOS. UNICOS is an operating system for Cray machines derived from AT&T Unix System V. It supports the restartable action paradigm by providing the programmer with primitives to checkpoint and restart processes. These primitives are provided in the form of two new system calls: `chkpnt` and `restart`. `chkpnt` is used to checkpoint a process; it creates a *restart* file containing the information needed to restore the process to execution at a later time. `restart` accepts a restart file and restores the process to the stored state. In addition to providing these new system calls, UNICOS defines a new Unix signal called *SIGRECOVERY* that is sent to a restarted process. This signal may be fielded by the process and recovery code executed by the signal handler.

Although an important building block, the restartable action paradigm is directly applicable to only a small class of mainly sequential applications. Moreover, even for these applications, coarse-grained checkpointing facilities such as those provided by UNICOS are sometimes less than optimal since they force the programmer to save an entire process state even if only a portion is needed. In FT-SR, the granularity of access to stable storage is controlled by the programmer, thereby allowing the facility to be tuned to the needs of the particular application. Of course, coarse-grained checkpointing can still be provided in the form of library routines if desired.

Argus. Argus is an integrated programming language and system that supports the object/action programming paradigm. Objects in Argus are called guardians and export operations called handlers. The *state* of a guardian is encoded by its variables, which may be *stable* or *volatile*. As the name implies, stable variables are stored in stable storage and survive failures, while volatile variables are lost when failures occur. A call to a handler results in the creation of a process to handle the call. This process may modify the variables of the guardian and invoke handlers on other guardians. Since the guardians may be on different machines, handler invocations may cross machine boundaries. An action in Argus can be thought of as a process that attempts to transform the state of one or more guardians from an initial state to a final state, with any number of intermediate state changes. All actions are atomic and complete by either committing or aborting. An action may also be aborted if a failure occurs while it is in progress. When an action aborts, a failure is signaled that can be fielded by a programmer-supplied signal handler associated with the action. When a failed processor recovers, the Argus runtime first recreates the guardians that were executing when the failure occurred. The system then restores the values of the stable variables from stable storage and executes the specified recovery code, if any. The guardian is ready to accept new invocations as soon as the recovery code completes.

As shown in Figure 9, languages like Argus that realize the object/action model often end up implementing an entire range of abstractions as part of the runtime support. From the perspective of the user, however, these intermediate abstractions are hidden and therefore not available to construct applications for which the object/action model is not directly suited. In contrast, the philosophy behind FT-SR is to build in only the common foundation, thereby allowing the user to construct whatever higher-level abstractions are most appropriate for a given application. Of course, as with checkpointing, those that are most commonly used can be provided as library routines if desired.

7 Concluding Remarks

A distributed programming language designed to support the construction of fault-tolerant systems must be flexible enough to allow a variety of structuring and redundancy techniques. FT-SR has been designed to be such a language by incorporating facilities targeted at supporting the various programming paradigms that have been proposed for such systems. These include support for encapsulation based on SR resources, synchronous and asynchronous failure notification, resource replication with consistent invocation ordering, and recovery. The logical basis of the language design is a programming model centered around the notion of fail-stop atomic objects.

Future work will concentrate on using FT-SR to construct a number of different prototype systems. This process will be used to gain experience with the language that can be used to refine and expand the design of the language. This will also help us identify and implement the library support necessary to simply the task of constructing such systems. Among the additional issues that we expect to address are the expansion of our failure-handling mechanisms into a general exception handling scheme oriented towards the specific nature and requirements of distributed programming languages, and the incorporation of provisions for real-time computing.

References

[AD76] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of Second International Conference on Software Engg.*, pages 562–570, October 1976.

[AO93] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language*. The Benjamin/Cummings Publishing Company, 1993.

[AOC+88] Gregory R. Andrews, Ronald A. Olsson, Michael A. Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purdin, and Gregg Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[Bal91] Henri E. Bal. A comparative study of five parallel programming languages. In *Proceedings of EurOpen Conference on Open Distributed Systems*, May 1991.

[BD88] A. Burns and G. Davies. Pascal-FC: A language for teaching concurrent programming. *ACM SIGPLAN Notices*, 23(1):58–66, January 1988.

[BH75] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–206, June 1975.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, 1987.

[BMZ92] Peter A. Buhr, Hamish I. MacDonald, and C. Robert Zarnke. Synchronous and asynchronous handling of abnormal events in the μSystem. *Software—Practice and Experience*, 22(9):735–776, September 1992.

[BSS91] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[CAS85] Flaviu Cristian, Houtan Aghili, and Ray Strong. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Digest of Papers, The Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206. IEEE Computer Society, June 1985.

[CGR88] R.F. Cmelik, N.H. Gehani, and W. D. Roome. Fault tolerant concurrent C: A tool for writing fault tolerant distributed programs. In *Digest of Papers, The Eighteenth International Symposium on Fault-Tolerant Computing*, pages 55–61. IEEE Computer Society, IEEE Computer Society Press, June 1988.

[CM84] Jo-Mei Chang and N.F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[Coo80] Robert P. Cook. *MOD—a language for distributed programming. *IEEE Transactions on Software Engineering*, SE-6(6):563–571, November 1980.

[Coo85] Eric C. Cooper. Replicated distributed programs. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 63–78. ACM SIGOPS, December 1985.

[Cri91] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[EFH82] C.S. Ellis, J.A. Feldman, and J.E. Heliotis. Language constructs and support systems for distributed computing. In *ACM Symposium on Principles of Distributed Computing*, pages 1–9. ACM SIGACT-SIGOPS, August 1982.

[GMS89] Hector Garcia-Molina and Annemarie Spauster. Message ordering in a multicast environment. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 354–361, June 1989.

[GR89] N. H. Gehani and W.D. Roome. *The Concurrent C Programming Language*. Silicon Press, Summit, NJ, 1989.

[Gra79] James N. Gray. Notes on data base operating systems. In R. Bayer, R.M. Graham, and G. Seegmuller, editors, *Operating Systems, An Advanced Course*, chapter 3.F, pages 393–481. Springer-Verlag, 1979.

[Gra86] James N. Gray. An approach to decentralized computer systems. *IEEE Transactions on Software Engineering*, SE-12(6):684–692, June 1986.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[HW87] Maurice P. Herlihy and Jeannette M. Wing. Avalon: Language support for reliable distributed systems. In *Digest of Papers, The Seventeenth International Symposium on Fault-Tolerant Computing*, pages 89–94. IEEE Computer Society, IEEE Computer Society Press, July 1987.

[KK89] Brent A. Kingsbury and John T. Kline. Job and process recovery in a UNIX-based operating system. In *Proceedings of the 1989 Winter USENIX Technical Conference*, pages 355–364, 1989.

[KTHB89] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, and Henri E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[KU87] John C. Knight and John I. A. Urquhart. On the implementation and use of Ada on fault-tolerant distributed systems. *IEEE Transactions on Software Engineering*, SE-13(5):553–563, May 1987.

[Lam81] Butler W. Lampson. Atomic transactions. In B.W. Lampson, M. Paul, and H.J. Seigert, editors, *Distributed Systems—Architecture and Implementation*, chapter 11, pages 246–265. Springer-Verlag, 1981. Originally vol. 105 of Lecture Notes in Computer Science.

[Lap91] Jean-Claude Laprie. *Dependability: Basic Concepts and Terminology*, volume 4 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, 1991.

[Lis85] Barbara Liskov. The Argus language and system. In M. Paul and H.J. Siegert, editors, *Distributed Systems: Methods and Tools for Specification, Lecture Notes in Computer Science, Volume 190*, chapter 7, pages 343–430. Springer-Verlag, Berlin, 1985.

[LL81] C.-M. Li and M.T. Liu. Dislang: A distributed programming language/system. In *Proceedings of the 2nd International Conference on Distributed Computing Systems*, pages 162–172, 1981.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[LW85] Richard J. LeBlanc and C. Thomas Wilkes. Systems programming with objects and actions. In *The 5th International Conference on Distributed Computing Systems*, pages 132–139, Denver, Colorado, May 1985. IEEE Computer Society.

[Mad86] Hari Madduri. Fault-tolerant distributed computing. *Scientific Honeyweller*, Winter 1986-87:1–10, 1986.

[MS92] Shivakant Mishra and Richard D. Schlichting. Abstractions for constructing dependable distributed systems. Technical Report TR 92-19, Department of Computer Science, University of Arizona, Tucson, AZ 85721, 1992.

[MSMA90] P.M. Melliar-Smith, Louise E. Moser, and Vivek Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

[PS88] Fabio Panzieri and Santosh K. Shrivastava. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Transactions on Software Engineering*, 14(1):30–37, January 1988.

[PVB+88] D. Powell, P. Verissimo, G. Bonn, F. Waeselynck, and D. Seaton. The Delta-4 approach to dependability in open distributed computing systems. In *Digest of Papers, The Eighteenth International Symposium on Fault-Tolerant Computing*, pages 246–251, June 1988.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SCP91] Richard D. Schlichting, Flaviu Cristian, and Titus D. M. Purdin. A linguistic approach to failure-handling in distributed systems. In Algirdas Avižienis and Jean-Claude Laprie, editors, *Dependable Computing and Fault-Tolerant Systems, Vol. 4: Dependable Computing for Critical Applications*, pages 387–409. Springer-Verlag, Wien and New York, 1991.

[SDD+85] Alfred Z. Spector, Dean Daniels, Daniel Duchamp, Jeffrey L. Eppinger, and Randy Pausch. Distributed transactions for reliable systems. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 127–146, 1985.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *IEEE Transactions on Computing Systems*, 1(3):222–238, August 1983.

[Wei89] William E. Weihl. Using transactions in distributed applications. In Sape Mullender, editor, *Distributed Systems*, pages 215–235. Addison-Wesley Publishing Company, ACM Press, New York, New York, 1989.

Appendix A: The SR Distributed Programming Language

An SR program consists of one or more *resources*. These resources can be thought of as patterns from which resource instances are created dynamically. Each resource is composed of two parts: an interface portion which specifies the interface of the resource and a *body*, which contains the code to implement the abstract object. The specification portion contains descriptions of objects that are to be exported from this resource—made available for use within other resources—as well as the names of resources whose objects are to be imported. Of primary importance are the declaration of *operations*—actions implemented by sequences of statements that can be invoked. These declarations specify the interface of those operations that are available for invocation from other resources. For example,

```
op example1(var x: int; val y: bool)
```

declares an operation, `example1`, that takes as arguments an integer `x` that is passed with copy-in/copy-out (**var**) semantics and a boolean `y` that is copy-in only (**val**). Result parameters (**res**) are also supported, as are operations with return values.

The declaration section in the resource body together with its specification define the objects that are global to the resource, i.e., accessible to any process within the resource. All of the usual types and constructors are provided. In addition, there are *capability variables*. Such a variable functions either as a pointer to all operations in a resource instance (a *resource capability*), or as a pointer to a specific operation within an instance (an *operation capability*). A variable declared as a resource capability is given a value when a resource instance is created, while an operation capability is given a value by assigning it the name of an operation or from another capability variable. Once it has a value, such variables can be used to invoke referenced operation(s), as described later.

The resource instances comprising a given program may be distributed over multiple *virtual machines*, which are abstract processors that are mapped to physical machines in the network. A resource instance is created and placed on a virtual machine using the following:

```
res_cap := create res_name(arguments) on virtual_machine_cap
```

Execution of this statement creates an instance of the resource `res_name` on the virtual machine specified by the virtual machine capability `virtual_machine_cap` and assigns a capability to the newly created resource to the capability variable `res_cap`.

An operation is an entry into a resource. An SR operation has a name, and can have parameters and return a result. There are two different ways to implement an operation: as a *proc* or as an alternative in an *input* statement. A proc is a section of code whose format resembles that of a conventional procedure:

```
proc opname(parameters) returns result
  op_body
end
```

The operation body `op_body` consists of declarations and statements. Like a procedure, the declarations define objects that are local to the operation `opname`. Unlike a procedure, though, a new process is created, at least conceptually, each time `opname` is invoked. It is possible to get standard procedure-like semantics, however, depending on how the proc is invoked (see below). The process terminates when (if) either its statement list terminates or a **return** is executed.

An operation can also be implemented as an alternative of an input statement. An input statement implementing a collection of operations $opname_1$, $opname_2$, . . ., $opname_n$ has the following form:

```
in opname₁(parameters) -> op_body₁
□ opname₂(parameters) -> op_body₂
  ...
□ opnameₙ(parameters) -> op_bodyₙ
ni
```

A process executing an input statement is delayed until there is at least one alternative $opname_i$ for which there is a pending invocation. When this occurs, one such alternative is selected non-deterministically, the oldest pending invocation for the chosen alternative is selected, and the corresponding statement list is executed. The input statement terminates when the chosen alternative terminates.

An operation is invoked explicitly using a **call** or **send** statement, or is implicitly called by its appearance in an expression. The explicit invocation statements are written as

```
call op_denotation(arguments)
send op_denotation(arguments)
```

where the operation is denoted by a capability variable or by the operation name if the statement is in the operation's scope. An operation can be restricted to being invoked only by a call or a send by appending a {**call**} or {**send**} operation restrictor to the declaration of the operation.

Execution of a **call** terminates once the operation has been executed and a result, if any, returned. Its execution is thus synchronous with respect to the operation execution. Execution of a **send** statement is, on the other hand, asynchronous: a **send** terminates when the target process has been created (if a proc), or when the arguments have been queued for the process implementing the operation (if an input statement). Thus, the effects of executing the various combinations of **send**/**call** and **proc**/**in** are described by the following table.

Invocation	*Implementation*	*Effect*
call	**proc**	procedure call
send	**proc**	process creation
call	**in**	rendezvous
send	**in**	asynchronous message passing

To illustrate how the individual pieces of the language fit together, consider the implementation of a bounded buffer shown in Figure 10. Two operations are exported from this resource: `deposit` and `fetch`; `deposit` places a value in the next available slot if one exists, while `fetch` returns the oldest value from the buffer. A depositing process is delayed should the buffer be full. Similarly, a fetching process is delayed whenever the buffer is empty. Note also that the resource has a parameter `size`; its value determines the number of slots in the buffer. The use of resource parameters in this way allows instances to be created from the same pattern, yet still vary to a certain degree. Finally, note the single input statement to implement both the deposit and fetch operations, and the use of a send statement in the initialization code to initiate the main (parameterless) proc `buff_loop`. Creating a process in this manner is so common that the keyword **process** can be used instead of **proc** as an abbreviation for the **send** in the resource initialization code and corresponding **op** declaration.

```
resource buffer
    op fetch() returns value:  int
    op deposit(val newvalue:  int)
body buffer(size:  int)
    var first, last:  int := 0, 0
    var slot[0:size - 1]:  int

    initial
        send buff_loop()
    end

    proc buff_loop()
        do true ->
            in deposit(newvalue) and first != (last + 1) % size ->
                slot[last] := newvalue
                last := (last + 1) % size
            □ fetch() returns value and first != last ->
                value := slot[first]
                first := (first + 1) % size
            ni
        od
    end
end
```

Figure 10: Bounded buffer resource

Appendix B: Data Manager Implementation

Here, we present the rest of the **procs** exported by the data manager described in Section 4. As outlined there, the data manager keeps track of all in-progress transactions in a status table `transStatus`.

```
proc startTransaction(tid, dataAddrs, numDataItems)
    var t:  int # index into the status table

    P(statusTableMutex);
    # find an empty slot t in the statusTable
        ...
    statusTable[t].transStatus := 'A' # mark transaction as active

    V(statusTableMutex);
    statusTable[t].tid := tid

    # acquire locks on data items
    lockManager.lock(tid, dataAddrs)

    statusTable[t].memCopy := new(dataArray)
    statusTable[t].numItems := numDataItems
    fa i := 1 to numDataItems ->
        statusTable[t].dataAddrs[i] := dataAddrs[i]
        ss.read(dataAddrs[i], sizeof(int), currentPointer)
        statusTable[t].currentPointers[i] := currentPointer
        ss.read(dataAddrs[i]+currentPointer, sizeof(data), statusTable[t].memCopy[i])
    af

    # write status table entry onto stable storage
    ss.write(statusTable + t*sizeof(transInfoRec), sizeof(trans), statusTable[t])
end startTransaction
```

Figure 11: `startTransaction` operation

Figure 11 shows `startTransaction`. The **proc** first finds an empty slot in the `statusTable`, i.e. a slot with a `transStatus` of 'E', and marks it as being actively used ('A') by transaction `tid`. The lock manager is then invoked to acquire locks on the data items. The status table entry is updated next; specifically, the address of the data array is assigned to `memCopy` and the `numItems` field is initialized. Information concerning each data item is then stored after being retrieved from stable storage if necessary; this information includes the address of the data item, its value, and its `currentPointer`. Finally, the appropriate status table entry on stable storage is updated. Once a transaction is started, the data items it uses may be accessed and modified using the `read` and `write` operations shown in Figure 12. These operations use the copy of the data items in volatile memory.

The `prepareToCommit` and `commit` operations are shown in Figure 13. `prepareToCommit` is invoked when the transaction manager decides to commit the transaction; it writes all the data items from the copy in volatile storage to the "non-current" copy in stable storage, and then discards the copies in volatile memory. The `commit` operation commits the modifications by changing the offset indicators of the appropriate data items in stable storage to point to the new version written by `prepareToCommit`. Following this, the status of the transaction is changed to done ('D') in both the volatile and stable storage versions. The data items are then unlocked. Finally, the transaction status is changed to empty ('E'), with the change being reflected onto stable storage as well. Since the transaction manager that co-ordinates the various data managers may re-issue commits when recovering from a failure, the `commit` operation may be re-executed in part or in total an arbitrary number of times given inopportune failures. Our implementation takes this into account by constructing this operation

```
proc read(tid, dataAddrs, data, numDataItems);
    # search transaction table for entry for tid.  let t be index of entry
        ...

    fa i := 1 to numDataItems ->
        j = 1;
        do (statusTable[t].dataAddrs[j] != dataAddrs[i]) ->
            j++
        od
        data[i] = statusTable[t].memCopy[j]
    af
end read


proc write(tid, dataAddrs, data, numDataItems);
    # search transaction table for entry for tid.  let t be index of entry
        ...

    fa i := 1 to numDataItems ->
        j = 1;
        do (statusTable[t].dataAddrs[j] != dataAddrs[i]) ->
            j++
        od
        statusTable[t].memCopy[j] = data[i]
    af
end write
```

Figure 12: `read` and `write` operations

as a restartable action.

Figure 14 shows the `abort` operation, which is invoked when the transaction manager decides to abort the transaction. `abort` simply discards the copies of the data items in volatile memory and changes the status of the transaction to empty ('E') in both the volatile and stable storage versions.

```
proc prepareToCommit(tid)
    var t:  int # index into the status table

    # search transaction table for entry for tid.  let t be index of entry
        ...

    # write modified objects to the "non-current" copy
    fa i := 1 to statusTable[t].numDataItems ->
        ss.write(statusTable[t].dataAddrs[i] + (statusTable[t].currentPointers[i] mod 2 + 1),
                sizeof(data), statusTable[t].memCopy[i])
    af
    free(statusTable[t].memCopy)
end prepareToCommit


proc commit(tid)
    var t:  int # index into the status table

    # search transaction table for entry for tid, let t be index of entry
        ...
    # if entry cannot be found, return---transaction has committed already
        ...

    if statusTable[t].transStatus = 'A' -> # transaction hasn't committed yet
        # replace current pointers of data items by new current pointers
        fa i := 1 to statusTable[t].numDataItems ->
            ss.write(statusTable[t].dataAddrs[i], sizeof(int),
                (currentPointers mod (sizeof(data)+1) + 1))
        af
        statusTable[t].transStatus := 'D' # mark transaction as done
    fi
    if statusTable[t].transStatus = 'D' -> # cleanup
        ss.write(statusTable + t*sizeof(transInfoRec), sizeof(trans), statusTable[t])
        lockManager.unlock(tid, statusTable[t].dataAddrs)
        statusTable[t].transStatus := 'E' # mark table slot as being empty
        ss.write(statusTable + t*sizeof(transInfoRec), sizeof(trans), statusTable[t])
    fi
end commit
```

Figure 13: `prepareTocommit` and `commit` operations

```
proc abort(tid)
    var t:  int # index into the status table

    # search transaction table for entry for tid.  let t be index of entry
        ...

    # free volatile copy of data items
    free(statusTable[t].memCopy)

    # change transaction status to empty
    statusTable[t].transStatus := 'E'
    ss.write(statusTable + t*sizeof(transInfoRec), sizeof(trans), statusTable[t])
end abort
```

Figure 14: `abort` operation