A Foundation for Adaptive Fault Tolerance in Software

K. Whisnant, Z. Kalbarczyk, and R.K. Iyer
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
1308 W. Main St.; Urbana, IL 61801
E-mail: {kwhisnan,kalbar,iyer}@crhc.uiuc.edu

Abstract

Software requirements often change during the operational lifetime of deployed systems. To accommodate requirements not conceived during design time, the system must be able to adapt its functionality and behavior. This paper examines a formal model for reconfigurable software processes that permits adaptive fault tolerance by adding or removing specific fault tolerance techniques during runtime. A distributed software-implemented fault tolerance (SIFT) environment for managing user applications has been implemented using ARMOR processes that conform to the formal model of reconfigurability. Because ARMOR processes are reconfigurable, they can tailor the fault tolerance services that they provide to themselves and to the user applications. We describe two fault tolerance techniques—microcheckpointing and assertion checking—that have been incorporated into ARMOR process via reconfigurations to the original ARMOR design. Experimental evaluations of the SIFT environment on a testbed cluster at the Jet Propulsion Laboratory demonstrate the effectiveness of these two fault tolerance techniques in limiting data error propagation among the ARMOR processes. These experiments validate the concept of using an underlying reconfigurable process architecture as the basis for implementing replaceable error detection and recovery services.

1 Introduction

Computing systems traditionally have been designed according to fixed specifications, and the resulting implementations often only incorporate those features necessary to satisfy the initial design requirements. As computers become more pervasive, user expectations begin to change. Users, however, are typically reluctant to frequently replace their existing systems with new hardware or software installations. But without upgrading these systems, their installations risk becoming unable to provide the desired level of service in an operating environment that changes over time. Ideally, the system should be able to adapt autonomously to a changing environment. For this to happen, the system must be designed to accommodate changes that are often unanticipated at design time.

This paper presents a formal system model for constructing distributed software that is generally reconfigurable (i.e., virtually any aspect of its functionality can change by adding, removing, or replacing software modules). The reconfigurable architecture creates a foundation for achieving adaptive fault tolerance, which is the ability of software to adjust its detection and recovery services to meet application dependability requirements. The concepts presented in this paper have been applied to ARMOR processes (Adaptive Reconfigurable Mobile Objects of Recovery) that form a software-implemented fault tolerance (SIFT) environment for managing third-party user applications [5]. There are two primary benefits of having the SIFT environment support adaptive fault tolerance:

1. **Application fault tolerance.** The services provided by the SIFT environment to the application can adapt to changing application requirements.

2. **ARMOR fault tolerance.** Additional fault tolerance techniques can be incorporated directly into the ARMOR processes that enhance the ability of the ARMOR processes to recover from their own errors. These techniques form an error detection and recovery hierarchy [1] that permits recovery from single failures as well as several multiple-failure scenarios [10].

The latter point illustrates how processes designed around the ARMOR architecture can be reconfigured to incorporate detection and recovery techniques not conceived at design time into running ARMOR processes. Of course, support must exist to detect the conditions under which an adaptation should occur and to determine the appropriate techniques to be brought online[1]. The ARMOR process architecture provides not only the underlying mechanisms

[1]Other issues pertinent to adaptivity include consistency across system changes, monitoring the execution environment, coordinating adaptations involving several components, and the cost-effectiveness of adaptation [9].

for performing dynamic adaptations, but also the ability to verify the correctness of the reconfigured system [12] with respect to certain criteria. Furthermore, this adaptivity is not limited to fault tolerance alone: any functional aspect of the ARMOR processes can be reconfigured by virtue of the ARMOR process architecture.

Most other adaptive fault tolerance approaches design software that can operate in one of several modes, with each mode offering a particular level of dependability. The job of the adaptive infrastructure is to transition between the modes at the appropriate time. Gong and Goldberg describe an algorithm that switches from a primary/backup execution mode to an actively replicated execution mode to tolerate a wider set of errors [4]. This system is adaptable insofar as it can change between two predefined execution modes. The AQuA middleware provides several voting strategies to use in replicating CORBA objects [3], and the application expresses its dependability requirements through quality of service objects [13]. Again, adaptivity is limited to the voting strategies implemented by the middleware. Other work formally represents a software system as a model, and adaptations at the model level are translated to changes in the software's executable components [6].

The ARMOR process architecture permits virtually any aspect of a program's behavior to be reconfigured during runtime, which lays the foundation for adaptivity that is not constrained to predetermined execution modes. The benefits of the reconfigurable ARMOR process architecture are demonstrated through experiments conducted on an ARMOR-based SIFT environment implemented for the Remote Exploration and Experimentation (REE) project at the Jet Propulsion Laboratory. Two fault tolerance techniques—microcheckpointing and assertion checks—were integrated into the ARMOR processes through reconfigurations on the original SIFT environment. These additional techniques were shown to make the SIFT environment more resilient to its own errors, particularly errors that corrupt the internal ARMOR data structures.

2 Formal Model of ARMOR Processes

This section formally introduces the structure and runtime behavior of reconfigurable ARMOR processes. Code and state in ARMOR processes are partitioned into replaceable modules. Threads execute by indirectly invoking code blocks in the process, and this level of indirection facilitates reconfiguring the runtime behavior of ARMOR processes. The formal model has been used to build a SIFT environment for managing user applications. Details of this ARMOR-based SIFT environment are given in the next section, followed by examples of how the SIFT environment can be reconfigured with additional fault tolerance support.

An ARMOR process is specified formally by a triple $(\mathcal{C}, \mathcal{V}, \mathcal{T})$:

- $\mathcal{C}$ is the set of code blocks in the system. A code block performs a computation triggered by events called *operations*.
- $\mathcal{V}$ is the set of state variables[2] in the system. Variables are only accessed through executing code blocks.
- $\mathcal{T}$ is the set of threads in the system. Threads execute by sequentially invoking code blocks in the system through operations, which bring about state changes by manipulating the system variables in $\mathcal{V}$.

The grouping of code blocks and variables into *elements* defines the structure of an ARMOR process, and the execution of code blocks within threads specifies the runtime behavior of the ARMOR process. By changing the set of code blocks invoked during the course of a thread's execution, the behavior of the ARMOR process can be reconfigured. By reconfiguring code blocks that provide fault tolerance[3], the fault tolerance services can adapt to changing dependability requirements.

2.1 Elements

ARMOR state variables are partitioned into components called *elements*. Code blocks are placed in the elements containing the state variables manipulated by the code blocks. Given a predicate $Access(c, v)$ that is true if and only if code block $c \in \mathcal{C}$ reads or writes variable $v \in \mathcal{V}$, an element can be defined formally as follows.

Definition 1 (Element) *An element is a pair (C, V), where V is a set of variables and C is a set of code blocks that do not access variables other than those in V:*

$$\text{element } e \equiv (C, V), \quad C \subseteq \mathcal{C} \wedge (\forall c \in C, V \subseteq \mathcal{V}, v \in \mathcal{V} : Access(c, v) \implies v \in V).$$

Elements partition the ARMOR so that each variable is found in one and only one element, and the code blocks within an element cannot access state variables residing in other elements.

2.2 Operations

Code blocks in the ARMOR process are indirectly invoked through *operations*. One operation can invoke several code blocks as defined by a function $BindCode\colon \mathcal{P} \longrightarrow 2^{\mathcal{C}}$, where $\mathcal{P}$ is the alphabet of operations and $2^{\mathcal{C}}$ is the power set over all code blocks (i.e.,

[2]Where appropriate, the *name* of a variable $v \in \mathcal{V}$ (v.name) is distinguished from the *value* of the variable (v.val).

[3]Criteria can be developed from the formal model to determine whether a proposed reconfiguration can be considered safe with respect to the current configuration's runtime behavior [12]

$BindCode$ maps an operation to a set of code blocks). An operation executes by being *delivered* to all code blocks bound to the operation via the $BindCode$ function.

Definition 2 (Operation Delivery) *An operation $p \in \mathcal{P}$ is said to be* delivered *to a code block $c \in \mathcal{C}$, denoted $Deliver(p, c)$, by executing code block c. The delivery completes when code block c completes executing.*

Because each code block exists in one and only one element, delivery can be thought of as occurring with respect to elements as well. An operation *executes* by being delivered to all bound code blocks (elements) in the ARMOR process.

Definition 3 (Operation Execution) *An operation $p \in \mathcal{P}$ is said to have* executed*, denoted Execute(p), after it has been delivered to all code blocks bound to p:*

$$Execute(p) \equiv \forall c \in BindCode(p) : Deliver(p, c).$$

Execution completes only after all deliveries for p complete.

2.3 Threads

The runtime behavior of an ARMOR process is characterized by a set of threads, each of which serially executes a sequence of operations. Given $\mathcal{P}$, the set of all finite operation sequences is given by $\mathcal{P}^*$. A sequence can be specified by its constituent operations as $P = \langle p_0, p_1, p_2, \dots \rangle$. We denote that p is in sequence P by $p \in P$.

A thread $T \in \mathcal{T}$ is specified by a triple $T = (P, V, F)$:

- $P \in \mathcal{P}^*$ is a sequence of operations organized as a stack, also referred to as $T.\mathrm{ops}$. Threads execute by sequentially executing the operations on the stack, beginning with the head operation. To emphasize that operations are executed within a thread, the notations used in Definitions 2 and 3 are extended to $T.Deliver(p, c)$ and $T.Execute(p)$.
- $V \in \mathcal{V}_{thd}$ is a subset of *private thread variables* that are only accessible through thread T. $\mathcal{V}_{thd}$ is disjoint from the state variables $\mathcal{V}$. Although two threads can contain private variables with the same names, each thread maintains its own independent value. The set of private variables for thread T is also referred to as $T.\mathrm{vars}$.
- F is a data structure called a *frame stack*, which is used to preserve the contents of private thread variables across nested operation invocations, essentially providing scope to the variables in the thread. A nested operation invocation occurs when an operation pushes a sequence of operations onto $T.\mathrm{ops}$.

```
1  procedure ExecuteThread($T$) :
2      while $T.\mathrm{ops} \neq \emptyset$ do
3          $p := \mathbf{pop}(T.\mathrm{ops})$
4          $T.Execute(p)$
5      end do
```

Figure 1: Pseudo code for thread execution.

The pseudo code for a thread's execution is given in Figure 1. The **while** loop executes until the operation stack is exhausted. The head operation is popped from the stack and stored in variable p. Line 4 executes the operation. Note that because the statement $T.Execute(p)$ does not complete until all deliveries for p complete (i.e., until all code blocks bound to p have executed), the operations within a thread are executed in a strictly sequential order.

2.4 Delivery Actions

The code blocks that execute during an operation delivery cannot make arbitrary state changes to the ARMOR process. Their effects are confined to the current thread and the element to which the operation is delivered. Specifically, a single delivery of operation p in thread T to code block c (i.e., $T.Deliver(p, c)$) can perform any of the following actions:

1. State variables of element e (the element to which c belongs) can be read. $c.\mathrm{readVars}$ denotes the set of state variables read by c.
2. State variables of element e can be written by code block c, denoted by $c.\mathrm{writeVars}$.
3. Private variables within thread T can be read by code block c, denoted by $c.\mathrm{readThdVars}$.
4. Private variables within thread T can be written by code block c, denoted by $c.\mathrm{writeThdVar}$.
5. The code block can atomically push new operations onto T's operation stack. Let $c.\mathrm{pushOps}$ refer to the finite *set* of operation sequences that can be pushed while c executes. At most one of these sequences, however, is pushed when c executes (this allows the code block to push different sequences depending upon runtime conditions).
6. The code block can create new threads whose states are initialized from the private variables in the parent thread T and the state variables in e.

If $c.\mathrm{writeVars} \neq \emptyset$ and $c \in BindCode(p)$, then the delivery $T.Deliver(p, c)$ requires *write access* to the element containing code block c. Otherwise, the delivery $T.Delivery(p, c)$ only requires *read access* to the element.

Because several threads can attempt to deliver an operation to a given element at the same time, multiple-reader/single-writer locks are used to control access to an element on a per-delivery basis.

3 ARMOR-based SIFT Environment

We have built a distributed software-implemented fault tolerance (SIFT) environment around ARMOR processes that conforms to the formal model presented in the previous section. The SIFT environment provides fault tolerance services to user applications, such as crash detection, hang detection, process restart, and process migration. The elements that implement each of these services can be replaced with alternate element implementations to alter the application detection and recovery policies.

A configuration of the SIFT environment has been deployed to protect parallel spaceborne applications as part of the Remote Exploration and Experimentation (REE) project at the Jet Propulsion Laboratory. This section briefly outlines the capabilities of the SIFT environment. The next section describes how fault tolerance techniques can be incorporated directly into the ARMOR processes by virtue of their reconfigurable architecture, thus making the ARMOR processes in the SIFT environment resilient to their own failures. Extensive fault injection experiments have been performed to validate the ability of the SIFT environment to recover from both application and ARMOR failures [10].

3.1 REE Applications

In traditional spaceborne applications, onboard instruments collect and transmit raw data back to Earth for processing. The amount of science that can be done is clearly limited by the telemetry bandwidth to Earth. The REE project intends to analyze the data onboard with a cluster of commercial off-the-shelf (COTS) components. The COTS components are expected to experience a high rate of radiation-induced transient errors in space (ranging from one per day to several per hour), and downtime directly leads to the loss of scientific data. The SIFT environment protects the MPI-based [8] scientific applications that execute on the REE processing cluster. Our experimental evaluations have used a texture analysis program from the Mars Rover mission [2] and a program from the Orbiting Thermal Imaging Spectrometer mission, both of which represent applications that stand to benefit from REE technology.

3.2 ARMORS in the SIFT Environment

All ARMOR processes share a common set of elements that implement core functionality, including the ability to (1) communicate with the local daemon process (to be introduced later), (2) respond to heartbeats from the local daemon, and (3) capture ARMOR state. There are four kinds of ARMORs that play specific roles in the SIFT environment:

1. **Fault Tolerance Manager (FTM).** A single FTM executes on one of the nodes in the REE cluster. It is responsible for recovering from ARMOR failures, recovering from application failures, and recovering from node failures by migrating affecting processes to another node in the cluster. The FTM contains the three basic elements described above in addition to elements that (1) accept job submission requests, (2) track resource usage of nodes in the SIFT environment, (3) send "Are-you-alive?" messages to each daemon in the cluster to detect node failures, (4) install Execution ARMORs for a particular application, (5) recover from failed subordinate ARMORs, (6) recover from node failures, and (7) recover from application failures.

2. **Heartbeat ARMOR.** The Heartbeat ARMOR executes on a node separate from the FTM. Its sole responsibility is to detect and recover from failures in the FTM through the periodic polling for liveness.

3. **Daemons.** Each node in the REE cluster executes a daemon process. Daemons are the gateways for ARMOR-to-ARMOR communication, and they detect failures in the local ARMORs. In addition to the elements that provide core ARMOR functionality, each daemon contains elements to (1) install other ARMOR processes on the node, (2) communicate with local ARMORs, (3) cache location information about remote ARMORs, (4) route messages to remote ARMORs, (5) send "Are-you-alive?" inquiries to the local ARMORs, (6) detect crash failures in local ARMORs, (7) process "Are-you-alive?" inquiries from the FTM, and (8) notify the FTM to initiate recovery of failed local ARMORs.

4. **Execution ARMORs.** Each application process is directly overseen by a local Execution ARMOR. The Execution ARMOR detects application crash failures (abnormal termination) and application hangs. Hangs are detected by the absence of "I'm-alive" messages from the application called *progress indicators*. In addition to the elements that provide core ARMOR functionality, Execution ARMORs contain elements to (1) launch application processes, (2) detect crash failures in application processes, (3) handle progress indicator updates from the application, (4) notify the FTM if the application process fails.

An external computer called the Spacecraft Control Computer submits application jobs to the FTM for execution. The FTM installs Execution ARMORs on the appropriate number of nodes in the REE cluster and instructs the Execution ARMORs to launch the application. Upon completion, the Execution ARMORs notify the FTM, which in turn notifies the Spacecraft Control Computer. Figure 2 depicts the SIFT environment when managing two REE applications.

4 Adaptable Fault Tolerance

The previous section described the functional characteristics of the ARMOR-based SIFT environment whose primary responsibility is to protect MPI applications that execute on the REE cluster. The ARMORs also incorporate fault tolerance to protect themselves from failure. These additional fault tolerance services are incorporated into the ARMOR processes as reconfigurations to the ARMOR functionality presented in section 3.2. This section presents two fault tolerance techniques that have been integrated into the ARMOR processes of the SIFT environment:

1. Microcheckpointing, an incremental checkpointing technique that transparently checkpoints ARMOR state on an element-by-element basis.

2. Assertion checks, which are inserted dynamically into ARMOR processes before or after particular operation deliveries, thereby helping ensure data integrity within the ARMOR process.

4.1 Microcheckpointing

ARMOR processes contain state that must be restored whenever an ARMOR is recovered after a failure. An incremental checkpointing technique called *microcheckpointing* leverages the structured execution of ARMOR processes to capture state on an element-by-element basis. Because of microcheckpointing, the stateful ARMOR processes can be recovered after failure without losing state; thus, the application typically is unaware of ARMOR failures [10].

Recall that threads within ARMOR processes execute by sequentially delivering operations to elements. According to the properties outlined in section 2.4, each delivery $T.Deliver(p, e)$ can bring about a state change only in element e. The basic idea of microcheckpointing is to capture the state of element e after each operation delivery $T.Deliver(p, e)$ (for more details, see [11]).

Algorithm. Each ARMOR process contains a *checkpoint buffer* in memory that accumulates a temporary checkpoint of the ARMOR as threads execute. The ARMOR periodically commits the checkpoint buffer to stable storage to make the checkpoint permanent (i.e., a checkpoint from which a failed ARMOR process can be recovered). Updates to the checkpoint buffer are made after operation deliveries that write to an element. These updates occur while an element is exclusively held for write access during the operation delivery; thus, because of the architecture described in section 2, no other threads can modify element state while the state is being copied to the checkpoint buffer. Furthermore, the checkpoint buffer is partitioned into $n+t$ disjoint regions, where n is the number of elements in the ARMOR and t is the number of threads that are allowed to execute concurrently in the ARMOR.

Example. Figure 3(a) presents an example thread in which two operations p_1 and p_2 sequentially execute. In this example, operation p_1 is bound to elements e_0 and e_1, while operation p_2 is bound only to e_2. Given this configuration, the execution of this thread consists of the operation deliveries in Figure 3(b). This figure also shows the locking/unlocking of each element before/after delivery. Figure 3(c) shows the same thread with microcheckpointing incorporated into the execution sequence. Note that after each delivery—while the element is locked—both the state of the element and the state of the current thread are copied to the ARMOR's checkpoint buffer. Using the formalisms outline in section 2, updates to the checkpoint buffer can be shown to result in a consistent snapshot of the ARMOR state when the checkpoint buffer is committed to stable storage[4].

The reconfigurable ARMOR architecture permits the microcheckpointing post-delivery "hooks" to be inserted into the execution of a thread. In general, these hooks cause extra computations to be performed in the thread beyond the usual operation deliveries. In this particular case, the hooks are exploited to add incremental checkpointing to the ARMOR process. Because microcheckpointing is inserted within the framework of the ARMOR architecture (specifically, the thread execution framework shown in Figure 1) any ARMOR process can transparently take advantage of microcheckpointing provided that state of its elements can be copied to the checkpoint buffer.

4.2 Assertion checks

In addition to microcheckpointing, internal assertion checks were added to the ARMORs in the SIFT environment to help protect against data errors. These assertion checks can be inserted or removed during runtime, giving the ARMORs the ability to adapt the level of fault tolerance that protects themselves.

The idea behind dynamically-insertable assertion checks is to identify vulnerable operation deliveries that require extra checking to protect against data errors. For example, an engineer may discover after a system operates in the field for a period of time that incorrect inputs

[4]Details of this argument are omitted for brevity.

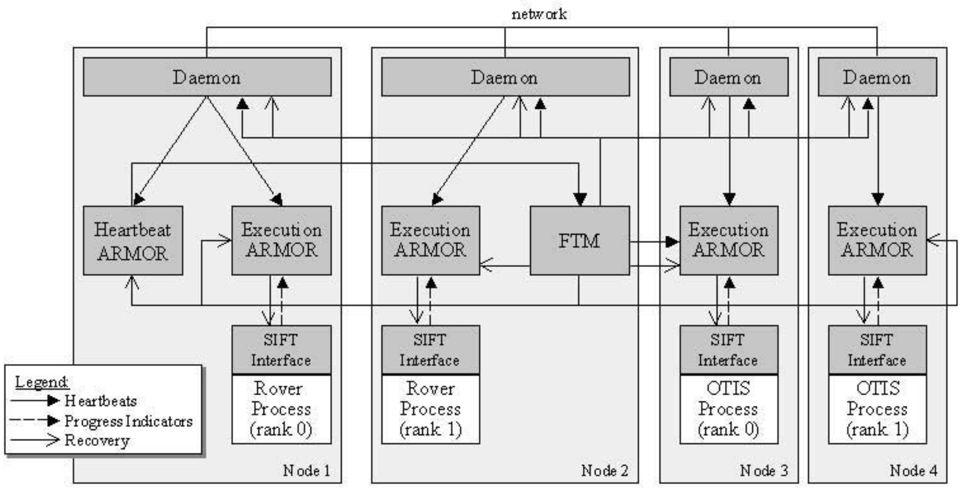


Figure 2: SIFT environment for executing two MPI applications concurrently.

to a particularly sensitive element lead to the ungraceful termination of the ARMOR process. To help prevent the element from being corrupted by the incorrect inputs, assertion checks can be added prior to the element's execution.

Consider, for example, an ARMOR process consisting of three elements e_0, e_1, e_2 and three operations p_0, p_1, p_2 bound to the elements as shown in Figure 4(a). A thread that executes the operation sequence $\langle p_0, p_1, p_2\rangle$ will invoke elements e_0, e_1, and e_2 in that order. In this example, an assertion check will be added to validate the inputs used during the delivery of operation p_1 to element e_1.

To insert the assertion-checking code, operation p_1 is rebound to a new element e_{glue} as shown in Figure 4(b). When the code block in e_{glue} executes during the delivery of operation p_1, the code block pushes the operation sequence $\langle \mathsf{OpAssert}, q\rangle$ onto the thread's operation stack. Operation OpAssert—the next operation to execute after p_1—invokes the assertion check encapsulated in element e_{assert}. This assertion check performs data integrity checks on the private thread variables used as input to element e_1.

If the assertion check fails, the element pushes the OpAssertFailed operation onto the thread's operation stack. Otherwise, the thread continues executing the next operation in the operation stack, which, in this case, is operation q. Operation q executes the code block in element e_1, the element previously bound to p_1 in the original configuration shown in Figure 4(b).

4.3 Impact of Microcheckpointing and Assertion Checks

The SIFT environment augmented with microcheckpointing and assertion checks has been subjected to extensive error injection experiments. Results from these experiments have shown that the combination of assertion checks and microcheckpointing is effective in limiting error propagation from data corruption that originate in the ARMOR processes [10]. The assertion checks detect invalid data, and the checkpoints captured through microcheckpointing allow the affected ARMOR process to rollback to a clean state. The effectiveness can be attributed to the fact that microcheckpointing does not capture incidental state changes made outside of a legitimate operation delivery. Because the microcheckpointing algorithm does not recognize that the state changes brought about by some errant operations, it does not copy the state of the affected element to the checkpoint buffer. If the error can be detected before the next legitimate write to the affected element, then a clean copy of the element's state exists in the ARMOR's tentative checkpoint buffer, thus permitting a successful recovery from the error.

To illustrate this, consider an example in which a thread delivers operation p to element e. During an error-free run, the delivery updates a state variable in element e. But because of an error, the operation errantly writes to another element e'. The ARMOR execution framework, however, does not realize that operation p has written to element e'. From the framework's standpoint, only element e has been modified; thus, only the state of element e is copied to the checkpoint buffer. The state of element e' prior to the corruption continues to exist in the checkpoint buffer.

As can be seen in the previous example, the ARMOR architecture adds information redundancy to a thread's execution. Specifically, the structured execution of threads limits the extent of state changes that are *expected* to occur

```
$T.Execute(p_1)$
$T.Execute(p_2)$
```

(a)

$\Longrightarrow$

```
lock $e_0$
$T.Deliver(p_1, e_0)$
unlock $e_0$
lock $e_1$
$T.Deliver(p_1, e_1)$
unlock $e_1$
lock $e_2$
$T.Deliver(p_2, e_2)$
unlock $e_2$
```

(b)

$\Longrightarrow$

```
lock $e_0$
$T.Deliver(p_1, e_0)$
ckptbuffer[$e_0$] $\leftarrow e_0$
ckptbuffer[$T$] $\leftarrow T$
unlock $e_0$
lock $e_1$
$T.Deliver(p_1, e_1)$
ckptbuffer[$e_1$] $\leftarrow e_1$
ckptbuffer[$T$] $\leftarrow T$
unlock $e_2$
lock $e_2$
$T.Deliver(p_2, e_2)$
ckptBuffer[$e_2$] $\leftarrow e_2$
ckptbuffer[$T$] $\leftarrow T$
unlock $e_2$
```

(c)

Figure 3: (a) Operations executing within thread T; (b) Individual deliveries for thread T; (c) Individual deliveries with microcheckpointing

for each operation delivery. Because of errors, however, the actual behavior can deviate from the expected behavior. In some cases, this can be detected immediately (e.g., if the element attempts to read from a thread variable that is not part of the operation delivery's input signature [12], an exception can be raised immediately). In the case of microcheckpointing, the microcheckpointing only captures state changes made to a legitimate element. Errant writes to another element or transient errors in other elements will not be reflected in the ARMOR's checkpoint until a legitimate write is made to the element. As long as the error is detected before the next legitimate write, then the propagation of the error can be prevented.

To evaluate the effectiveness of assertion checks and microcheckpointing in limiting error propagation, single-bit errors were injected into the FTM ARMOR's data structures residing on the heap. The remainder of this section summarizes these results (details appear in [10]).

Error Injections. Five elements with significant state in the FTM were targeted for error injection. One hundred runs were conducted for each element (each run consisted of the installation of the SIFT environment, the submission of an application job, the execution of the application, and the application completion notification from the FTM). In each run, a single error was injected into non-pointer data maintained by the target element. Non-pointer data were selected to maximize the likelihood of error propagation; corrupting pointers generally results in either a segmentation fault or bus alignment error when the FTM dereferences the corrupted pointer, causing an immediate crash.

The figure of merit from these experiments is the number of *failures*. A run failed if the application could not start because of the error, if the application did not complete execution because of the error, or if the FTM was not notified of a completed application because of the error. Results were also tabulated as to whether or not an assertion check detected the injected error. There were four possible outcomes for each run:

1. The error was not detected by an assertion check and did not result in a failure. In these runs, the application completes as if there were no error in the system.
2. The error was not detected by an assertion check but led to a failure. Because the FTM could not detect an incorrect condition in the system, no recovery was attempted.
3. The error was detected by an assertion check but led to a failure. In these runs, the error had already propagated outside the FTM or was written to the FTM's checkpoint before being detected. Rolling back the FTM's state in these circumstances failed to recover the system.
4. The error was detected by an assertion check and the system recovered to a clean state.

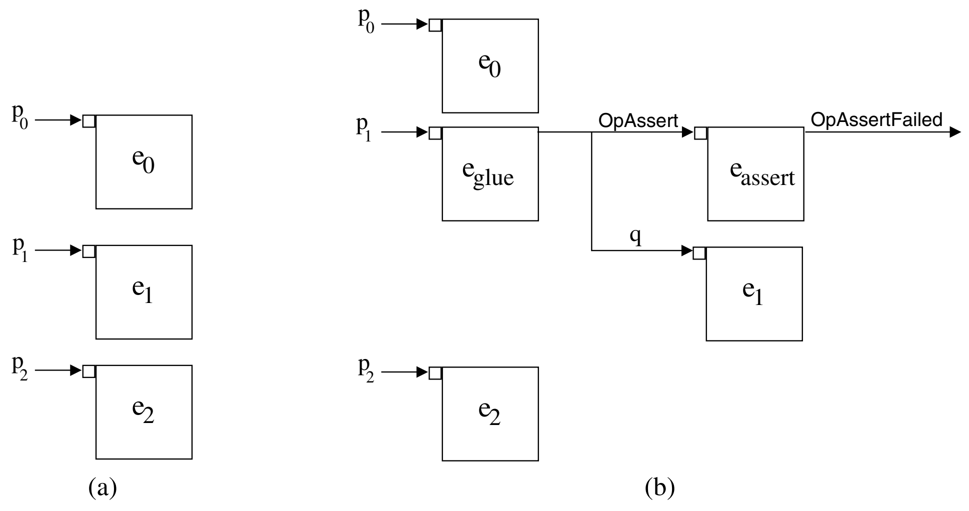


Figure 4: (a) Original ARMOR configuration; (b) ARMOR configuration with assertion check before $T.Deliver(p_1, e_1)$.

Table 1 summarizes the results for 500 runs (details of the target elements and other injection results can be found in [10]). The rightmost two columns represent the the total number of runs in which assertions detected the injected error. For example, assertions in the `mgr_armor_info` element detected 27 errors, and 19 of those errors were successfully recovered (this information is depicted by the Venn diagram to the right of the table).

Assertion checks coupled with microcheckpointing prevented failures in 58% of the cases (27 of 64 runs in which the assertion fired). In another set of experiments, recovery was not initiated after an assertion check detected an error (only the active thread was aborted). In all of these cases, the system failed. This demonstrates that the last column in Table 1 represent bona fide successes, not runs that would have succeeded with or without the assertion checks.

Both assertion checks and microcheckpointing represent reconfigurations to the original ARMOR processes that extend the functional behavior of the ARMORs with additional fault tolerance mechanisms. As new fault tolerance techniques are designed that leverage the ARMOR architecture, these can be integrated into existing ARMOR processes. Because these integrations can occur during runtime, the reconfigurable ARMOR architecture allows the SIFT environment to adapt to changing dependability requirements.

5 Conclusions and Future Work

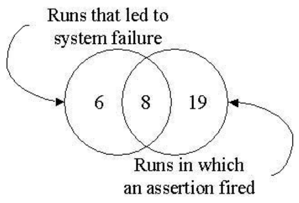
This paper has presented a formal model for constructing reconfigurable processes. The formal model partitions code and state variables into disjoint modules called elements. Elements are invoked indirectly through operations, and a thread's execution is modeled as sequential deliveries of operations to elements. Altering the bindings between operations and elements effectively reconfigures the runtime behavior of the process [12]. When the reconfigurability properties of the model are applied to fault tolerance concerns, the model provides a foundation for realizing adaptive fault tolerance.

The formal model for reconfigurability has been applied to a distributed SIFT environment for managing user applications around ARMOR processes [5]. Because the ARMOR processes are reconfigurable, the level of fault tolerance that they provide to the user application and to themselves can change according to user requirements. An initial configuration of the ARMOR-based SIFT environment has been reconfigured with support for microcheckpointing, an incremental checkpointing technique, and assertion checks that occur either before or after selected operation deliveries. The SIFT environment augmented with these two additional fault tolerance techniques hase been demonstrated on a testbed cluster at the Jet Propulsion Laboratory to protect the ARMOR processes and spaceborne scientific applications from errors in dynamic data [10].

Currently, ARMOR processes are being used to provide adaptive fault tolerance to very large real-time embedded computer systems. BTeV, an accelerator-based high energy physics project conducted at Fermilab [7], employs thousands of digital signal processors and thousands of general-purpose microprocessors organized in a processing hierarchy to collect noteworthy data from approximately 15 million subatomic particle interactions per second. Project requirements dictate that the system be highly reconfigurable from both a performance and dependability perspective—maximum performance must be delivered despite occasional failures in the computing resources. ARMOR pro-

Table 1: Efficiency of assertion checks in preventing failures.

Element	Failures Without Assertion Firing	Failures After Assertion Fires	Successful Recovery After Assertion Fires
`mgr_armor_info`	6	8	19
`exec_armor_info`	4	5	9
`app_param`	0	0	2
`mgr_app_detect`	0	0	4
`node_mgmt`	0	14	3
TOTAL	10	27	37


Runs that led to system failure
6
8
19
Runs in which an assertion fired

cesses encapsulate flexible detection and recovery policies in elements, and the reconfigurable underpinnings of the architecture allow the policies to adapt not only manually to changes requested by the operator, but also automatically to changes in the observed error profile.

Acknowledgments. This work was supported in part by NASA/JPL contract 961345, NSF ITR/AP 15917-S3, NSF CCR 00-86096 ITR, and NSF CCR 99-02026.

References

[1] S. Bagchi, B. Srinivasan, K. Whisnant, Z. Kalbarczyk, and R. Iyer. Hierarchical error detection in a SIFT environment. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):203–224, March/April 2000.

[2] F. Chen et al. Demonstration of the Remote Exploration and Experimentation (REE) fault-tolerant parallel-processing supercomputer for spacecraft onboard scientific data processing. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 367–372, 2000.

[3] M. Cukier et al. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the 17th Symposium on Reliable and Distributed Systems*, pages 245–253, 1998.

[4] L. Gong and J. Goldberg. Implementing adaptive fault-tolerant services for hybrid faults. Technical Report SRI-CSL-94-04, Computer Science Laboratory, SRI International, 1994.

[5] Z. Kalbarczyk, R.K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):560–579, June 1999.

[6] G. Karsai and J. Sztipanovits. A model-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May/June 1999.

[7] Fermi National Accelerator Laboratory. The BTeV project at Fermilab. http://www-btev.fnal.gov.

[8] Message Passing Interface Forum. MPI-2: Extensions to the message passing interface. http://www.mpi-forum.org/docs/mpi-20.ps, July 1997.

[9] P. Oreizy et al. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):46–53, May/June 1999.

[10] K. Whisnant, R.K. Iyer, P. Jones, R. Some, and D. Rennels. An experimental evaluation of the REE SIFT environment for spaceborne applications. In *Proceedings of the 2002 International Dependable Systems and Networks*, pages 585–595, June 2002.

[11] K. Whisnant, Z. Kalbarczyk, and R.K. Iyer. Micro-checkpointing: Checkpointing for multithreaded applications. In *Proceedings of the 6th IEEE International On-Line Testing Workshop*, July 2000.

[12] K. Whisnant, Z. Kalbarczyk, and R.K. Iyer. A system model for dynamically reconfigurable software. *IBM Systems Journal*, 42(1):45–59, April 2003.

[13] J. Zinky, D. Bakken, and R. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):55–73, 1997.