A system model for dynamically reconfigurable software

by K. Whisnant
Z. T. Kalbarczyk
R. K. Iyer

The ability to reconfigure software is useful for a variety of reasons, including adapting applications to changing environments, performing on-line software upgrades, and extending base application functionality with additional nonfunctional services. Reconfiguring distributed applications, however, can be difficult in practice because of the dependencies that exist among the processes in the system. This paper formally describes a model for capturing the structure and run-time behavior of a distributed system. The structure is defined by a set of elements containing the state variables in the system. The run-time behavior is defined by threads that execute atomic actions called operations. Operations invoke code blocks to bring about state changes in the system, and these state changes are confined to a single element and thread. By creating input/output signatures based upon the variable access patterns of the code blocks, dataflow dependencies among operations can be derived for a given configuration of the system. Proposed reconfigurations can be evaluated through off-line tests using the formal model to determine whether the new mapping of operations-to-code blocks disrupts existing dataflow dependencies in the system. System administrators—or software components that control adaptivity in autonomic systems—can use the results of these tests to gauge the impact of a proposed reconfiguration on the existing system. The system model presented in this paper underpins the design of reconfigurable ARMOR (Adaptive Reconfigurable Mobile Objects of Reliability) processes that provide flexible error detection and recovery services to user applications.

Reconfigurability provides the foundation upon which autonomic systems can adapt to their changing environments, which is useful for dynamically optimizing system functionality based upon the observed execution profile or for recovering from errors and failures without human intervention. Unfortunately, reconfiguring distributed systems is difficult in practice, since the processes are often interdependent. Arbitrarily changing the behavior of one process through reconfiguration may render other processes unusable. By formally expressing the dependencies among processes, both static and dynamic reconfigurations can be analyzed to determine whether they are compatible with the existing configuration.

This paper describes a model for formally capturing the structure and run-time behavior of a distributed system. The structure is defined by a set of elements containing the state variables in the system. The run-time behavior is defined by threads that execute atomic actions called *operations*. Operations invoke code blocks to bring about state changes in the sys-

tem, but these state changes are limited to a single element and thread. The variables accessed during the execution of an operation constitute the signature of the operation. The collective set of operation signatures in the system can be used to derive dataflow dependencies between operations and between elements.

Reconfiguring the system is modeled by changing the code blocks that are invoked by an operation. Given the current configuration of the system, a proposed reconfiguration can be subjected to an off-line test to determine whether any of the dataflow dependencies of the system change as a result of the reconfiguration. If there are changes to the dataflow dependencies, the user is warned that the proposed reconfiguration may not be safe. Ultimately, the user decides whether or not to apply a reconfiguration.[1]

We have employed the proposed system model in developing a software-implemented fault-tolerant (SIFT) environment based around ARMORs (Adaptive Reconfigurable Mobile Objects of Reliability) to provide error detection and recovery for distributed applications. To ensure that the SIFT environment is resilient to failures, dedicated fault-tolerant mechanisms were added by reconfiguring the ARMOR processes[2] that make up the SIFT environment. The system model and associated criteria for safe reconfigurations were applied to verify that the additional fault-tolerant mechanisms did not impact the SIFT functionality in unexpected ways.

System model

An abstract model that captures the structure and run-time execution of the system is used to analyze the effects of reconfiguration on the dependencies among the various components in the system. A system is specified 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[3] 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 $\mathcal{V}$.

Elements. State variables in the system 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:*

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

Elements partition the system 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.

Operations. Code blocks in the system are indirectly invoked through operations. One operation can invoke several code blocks as defined by a function *BindCode*: $\mathcal{P} \rightarrow 2^{\mathcal{C}}$, where $\mathcal{P}$ is the alphabet of operations and $2^{\mathcal{C}}$ is the power set over all code blocks (i.e., *BindCode* maps an operation to a set of code blocks). The notation "$p \rightarrow c$" is also used to denote that operation p is bound to code block c via the *BindCode* function. 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 system. Execution completes only after all deliveries for p complete.

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:*

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

Threads. The run-time behavior of the system is characterized by a set of threads, each of which se-

rially executes a sequence of operations. Given $\mathcal{P}$, the set of all operation sequences is given by $\mathcal{P}^*$. A sequence can be specified by its constituent operations as $P = \langle p_0, p_1, p_2, \ldots \rangle$. We denote p to be in sequence P by $p \in P$, and that p precedes q in sequence P by $p \prec_P q$.

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.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.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.ops.

The pseudocode for the execution of a thread 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 5 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.

Delivery actions. The code blocks that execute during an operation delivery cannot make arbitrary state changes to the system. 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.readVars denotes the set of state variables read by c.

Figure 1 Pseudocode for thread execution

```
1  procedure ExecuteThread(T) :
2      while T.ops≠∅ do
3          p := head(T.ops)
4          T.ops := tail(T.ops)
5          T.Execute(p)
6      end do
```

2. State variables of element e can be written by code block c, denoted by c.writeVars.
3. Private variables within thread T can be read by code block c, denoted by c.readThdVars.
4. Private variables within thread T can be written by code block c, denoted by c.writeThdVars.
5. The code block can atomically push new operations onto the operation stack of T. Let c.pushOps refer to the *set* of operation sequences that can be pushed while c executes. Only one of these sequences, however, is pushed when c executes (allows the code block to push different sequences depending upon run-time 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.

These six items constitute a signature of code block c, and the signature provides a succinct, black-box description of how the code block manipulates element state and thread state during run time. Programmers specify the signature as meta-data for each code block that they develop. The collective set of signatures for all code blocks in the system are used to derive dataflow dependencies among operations given the current *BindCode* mapping function.

Intrathread dependencies among operations. Because elements are allowed to read from private thread variables during an operation delivery, a particular operation delivery may be dependent upon deliveries earlier in the execution of the thread that wrote to thread variables. To better understand the dependencies among operations within a thread, the concept of an input/output signature is developed with respect to individual operations and sequences of operations as follows:

- An *input signature for an operation* describes the thread variables that are read by an operation when it executes.

- An *output signature for an operation* describes the thread variables that are written when an operation executes.
- An *input signature for an operation sequence* describes the thread variables whose values must be established before a sequence begins executing. These variables serve as inputs to the aggregate sequence of operations.
- An *output signature for an operation sequence* describes the thread variables that are the intended outputs of a sequence of operations. An operation sequence is pushed onto the operation stack of a thread with the express purpose of writing to the thread variables in the output signature of the sequence.

Input signatures. First, the thread variables used as input for operation delivery $T.Deliver(p, c)$ are considered. Define a function $DeliveryInputSig$: $\mathcal{P} \times \mathcal{C} \rightarrow 2^{\mathcal{V}_{thd}}$ as follows:

$$DeliveryInputSig(p, c) = c.\text{readThdVars} \cup \left(\left(\bigcup_{P \in c.\text{pushOps}} SeqInputSig(P) \right) - c.\text{writeThdVars} \right) \quad (1)$$

This definition states that the set of input variables used by $T.Deliver(p, c)$ not only includes c.readThdVars but also includes those thread variables used as input for the operation sequences pushed by code block c. The outermost parenthetical expression in Equation 1 includes the input variables of the sequence (denoted by $SeqInputSig(P)$, to be defined later), but excludes those variables used by the sequence that were written by c.[4] The intuition is that the input signature for $T.Deliver(p, c)$ should only include those variables whose values were defined *before* code block c executed.

In general, operation p can be delivered to several code blocks, and the composite input signature for operation p denoted by $ExecInputSig(p)$: $\mathcal{P} \rightarrow 2^{\mathcal{V}_{thd}}$ can be derived from Equation 1:

$$ExecInputSig(p) = \bigcup_{c \in BindCode(p)} DeliveryInputSig(p, c)$$

The input signature for a sequence P is slightly more complicated. If an operation $p \in P$ takes variable v as input, then v should not be part of the input signature for P if v was written by an operation that preceded p in P. Define $SeqInputSig$: $\mathcal{P}^* \rightarrow 2^{\mathcal{V}_{thd}}$ as follows:

$$SeqInputSig(P) = \{v \in \mathcal{V}_{thd} : \exists\, p \in P : v \in ExecInputSig(p) \wedge \forall q \prec_P p : v \notin ExecOutputSig(q)\}$$

Output signatures. As with input signatures, the output signature is defined first with respect to a single delivery $p \rightarrow c$ and then extended to account for all code blocks bound to p.

$$DeliveryOutputSig(p, c) = c.\text{writeThdVars} \cup \bigcup_{P \in c.\text{pushOps}} SeqOutputSig(P),$$

$$ExecOutputSig(p) = \bigcup_{c \in BindCode(p)} DeliveryOutputSig(p, c) \quad (2)$$

The delivery output signature in Equation 2 includes not only the variables written directly by c, but also the variables written by any sequence of operations pushed by c.

The output signature for an operation sequence P is defined, in general, by the programmer. The programmer who writes code block c pushes sequence P with the intent of producing specific outputs. The individual operations in P may write intermediate values as P executes, but these intermediate results ultimately will be discarded when P completes (see the discussion of the frame stack that follows). Discarding the intermediate values produced by P provides scope to the thread variables, and this is similar to saving or restoring register contents when entering or exiting a function in order to preserve the contents of registers not intended to store function outputs. Because the intended output of an operation sequence cannot be derived without semantic knowledge of the sequence, the output signature $SeqOutputSig(P)$ must be specified by the programmer.

Frame stack. Each thread contains a frame stack that implements the saving or restoring of private thread variables described above. When an operation sequence is pushed on the operation stack of the thread, the current values of all intermediate output variables for the sequence are saved. This set consists of all thread variables that will be written by the operations in the pushed sequence P, exclud-

ing those variables designated as the intended output of the sequence:

$$\bigcup_{p \in P} ExecOutputSig(p) - SeqOutputSig(P)$$

The variables stored in the top entry of the frame stack are restored when the operation sequence P completes.

Concurrency. Operations within a thread execute in a sequential order. Only one thread executes within an element (i.e., delivers operations to code blocks within an element) at any given time to ensure mutually exclusive access to the state variables of the element. This assumption treats each delivery as if it were writing to the element, but this requirement can be loosened if the deliveries can be tagged as "read-only" or "read/write," in which case a multiple-read, single-writer lock can be used to control access to the elements.

Applying the model to object-oriented programs. In object-oriented systems, objects are similar to elements in the sense that objects typically encapsulate state and member functions that operate on the state of the object. Each object, therefore, has an associated signature for each of its member functions. When these member functions are invoked within a thread, dataflow dependencies exist among the objects visited by the execution of a thread. This is the intuition behind the construction of the model presented in this section.

Elements, however, more generally represent sets of related objects. For example, a tree data structure may represent each node as an object, but the entire tree would most likely be encapsulated in an element. Keeping I/O signatures at the coarser granularity level of elements makes the dataflow dependability analysis more manageable than if per-object signatures were required.

Member function invocations, therefore, are modeled as operations being delivered to elements. Nested function invocations are split into multiple operation deliveries, since the model defines the execution of a thread as the sequential execution of operations. For example, consider a function X that performs some computation C_1, calls function Y, and performs an additional computation C_2 after Y returns. This execution trace is modeled as three operation deliveries: the execution of C_1, followed by the execution of Y, followed by the execution of C_2. Each of these operation deliveries represents an atomic state change to the system state.[5]

With programs explicitly designed around the system model presented in this paper, a structure is in place for extracting information to perform dataflow analysis on the individual operations that make up the execution of the system. Furthermore, the model allows the information to be gathered and analyzed in an automated fashion, given the I/O signatures of each individual operation. If the dataflow analysis framework is to be applied to object-oriented programs not designed with the proposed system model in mind, then the issue becomes one of how to extract the appropriate information needed for the dataflow analysis (e.g., how do objects pass information between each other within a thread of execution, what information is exchanged through each object invocation, and what are the effects of each object's invocation on system state). After gathering this information, the effects of a reconfiguration on the system can be determined. Obtaining such information may require manual input from the programmer or may require refactoring certain aspects of the application to more closely conform to a system model such as the one proposed in this paper to facilitate the automated dataflow analysis of proposed reconfigurations.

Reconfigurability

The sets of code blocks, elements, and operations are considered to be static, and reconfiguration occurs by either adding or removing a single operation binding (i.e., changing the *BindCode* function). The system transitions into a new configuration view whenever the *BindCode* function changes. A configuration view is denoted by $\mathbb{V}_i$, with configuration view $\mathbb{V}_{i+1}$ occurring immediately after $\mathbb{V}_i$. When necessary, the configuration view index will be specified along with the *BindCode* function (e.g., $BindCode_i(p)$ specifies the set of code blocks bound to p under configuration view $\mathbb{V}_i$).

Impact on thread state. The *BindCode* function determines the I/O signatures for operations and, therefore, establishes dependencies among operations within a thread of execution. Programmers design code blocks with an understanding of these dependencies in order to bring about a desired effect. For example, if operation p_k copies data from element e_1 to variable v, and operation p_{k+1} copies data from variable v to element e_2, data are transferred from one e_1 to e_2 by virtue of the I/O signatures of p_k and

p_{k+1}. If a reconfiguration—either adding or removing an operation binding—disrupts the data dependencies for future operations in the execution of a thread, then the reconfiguration is said to be unsafe.

Definition 4 (Safe Reconfiguration): *A reconfiguration from view* $\mathbb{V}_i$ *to* $\mathbb{V}_{i+1}$ *that affects the binding* $p \rightarrow c$, *where* $p \in \mathcal{P}$ *and* $c \in \mathcal{C}$, *is considered to be* safe with respect to thread state *if and only if, for all executed operation sequences* $P \in \mathcal{P}^*$ *in which* $p \in P$:

1. *All dataflow dependencies*[6] *between operations following* p *and operation* p *in view* $\mathbb{V}_i$ *continue to exist in view* $\mathbb{V}_{i+1}$.

2. *All dataflow dependencies between operations following* p *and operation preceding* p *in view* $\mathbb{V}_i$ *continue to exist in view* $\mathbb{V}_{i+1}$:

$$\forall P \in \mathcal{P}^* : p \in P \Rightarrow \forall q \preceq_P p, r \succ_P p:$$

$$r \text{ dataflow-dependent on } q \text{ in } \mathbb{V}_i \Rightarrow r \text{ dataflow-dependent on } q \text{ in } \mathbb{V}_{i+1}$$

System reconfigurations change the binding of operation p, either assigning a code block to p (causing an extra computation to be performed) or removing a code block from p (eliminating a computation from threads that execute in the system). A reconfiguration is unsafe only if it affects operations further downstream in the execution of the thread. The following examples illustrate safe reconfigurations in which dataflow dependencies are added, removed, or changed:

- A new binding $p \rightarrow c$ reads from some thread variable v, resulting in a new dataflow dependency that did not exist in the previous configuration. This reconfiguration is safe, since it does not disrupt the execution of operations further downstream.

- A removed binding $p \rightarrow c$ causes p to no longer read from thread variable v. Obviously, this reconfiguration destroys a dataflow dependency that previously existed, but the reconfiguration is not considered to be unsafe since this, in itself, does not affect operations further downstream in the execution of the thread.

  After the binding $p \rightarrow c$ is removed, it may be safe to remove the binding that wrote to v earlier in the execution of the thread. This is an example of how functionality often can be removed incrementally: the safe reconfiguration criteria help identify the correct order in which multiple bindings should be removed to bring the system safely to a desired configuration.

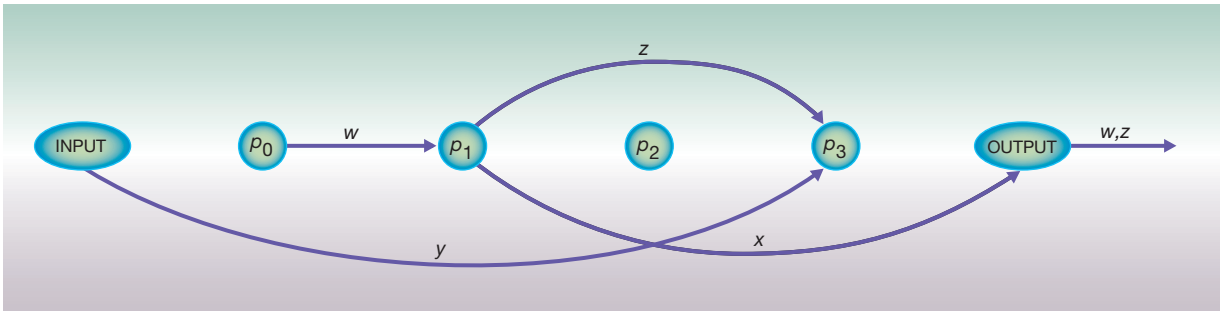
- A new binding $p \rightarrow c$ writes to thread variable v that is not currently read by any other operation further downstream in the execution of the thread. Although this lone reconfiguration does not immediately bring about a change in functionality, additional reconfigurations can be made so that operations further downstream read from thread variable v. As an alternative, a reconfiguration can be made so that a later operation pushes a new operation sequence onto the operation stack of the thread with the express purpose of performing some computation based upon the newly written value in variable v.

As the previous examples suggest, several reconfigurations can be made to remove or replace specific functionality in the system. The dataflow dependency analysis assists in deciding the proper order to apply such reconfigurations so that the correct behavior of the system is not disturbed. It should be clear, however, that the safe reconfiguration criteria presented in this section cannot guarantee consistency using dataflow dependency analysis alone. Definition 4 is a sufficient but not necessary condition for preserving consistency across reconfigurations. A semantic analysis of the system is required to make this consistency determination, and the semantics of a code block cannot be derived solely from the code block I/O signatures. Dataflow analysis, however, permits the programmer to understand the relationships between a new code block and the other code blocks in the system, simply by incorporating the signature of the new code block into the following analysis framework.

Dataflow analysis framework. The analysis framework and resulting criteria depend only upon the current system configuration (i.e., the $BindCode_i$ function and resulting I/O signatures) and the proposed configuration (i.e., $BindCode_{i+1}$). If the criteria indicate that the proposed reconfiguration may be unsafe, the user is notified of the existing dataflow dependency that would be broken by the reconfiguration. Other reconfigurations may be needed to compensate for the broken data dependency.

Figure 2, used throughout this section as a running example, shows data dependencies among operations in a sequence $P = \langle p_0, p_1, p_2, p_3 \rangle$ that was

Figure 2 Read/write dependencies among operations in sequence $p = \langle p_0, p_1 p_2, p_3 \rangle$



pushed onto the operation stack. The arcs between nodes indicate the dataflow dependencies between operations (e.g., operation p_1 writes to a variable $z \in \mathcal{V}_{thd}$, which is later read by p_3). Arcs emanating from input define the input signature for P (e.g., p_3 reads from variable y, whose value was not altered by either p_1 or p_2). Arcs that point to the output node represent the variables that are part of the output signature for sequence P. Variables that store intermediate values are shown to emanate from the output node, indicating that they will be restored when P completes. A dataflow dependency graph can be constructed completely for any sequence P given the I/O signatures for P and the individual operations in P.

The dataflow dependencies in Figure 2 can be altered by either adding or removing an operation binding. Since the analysis performed in both cases is similar, the discussion focuses on adding an operation binding $p \rightarrow c$, without loss of generality. Adding a binding $p \rightarrow c$ is considered to be safe if $DeliveryOutputSig(p, c) = \emptyset$, since thread variables are not written when p is delivered to c.

If $DeliveryOutputSig(p, c) \neq \emptyset$, then how the additional writes affect later operations in the thread must be examined. For illustrative purposes, suppose that the binding $p_2 \rightarrow c$ is being added to the example in Figure 2. The analysis decomposes the effect of the additional write into two cases:

1. How the additional write from $p_2 \rightarrow c$ affects later operations within P (intrasequence dependencies)
2. How the additional write from $p_2 \rightarrow c$ affects operations that follow execution of P in the thread (extrasequence dependencies)

Intrasequence dependencies. The first case examines how adding binding $p \rightarrow c$ affects the dataflow among operations within the sequence to which p belongs. If an operation p' that follows p in sequence P reads from a variable written by the delivery $T.Deliver(p, c)$, then p' is dependent upon the reconfiguration if no other operation between p and p' overwrote variable v. This condition is formally expressed in the following theorem (proofs are omitted for the sake of brevity).

Theorem 1 (Intrasequence Safety): *Given an operation sequence $P \in \mathcal{P}^*$ and an operation $p \in P$, the binding $p \rightarrow c$ can be safely added to the system only if any variables written by $p \rightarrow c$ do not overwrite the values in the variables expected by later operations within P. Formally, a safe configuration implies:*

$$\forall p' \succ_P p : V = (ExecInputSig(p') \cap DeliveryOutputSig(p, c)) \wedge V \neq \emptyset \Rightarrow \forall v \in V: \exists p'' \in P : p \prec_P p'' \prec_P p' \wedge v \in ExecOutputSig(p'').$$

In the following examples, the binding $p_2 \rightarrow c$ is added to the system described in Figure 2. The four cases differ with respect to the variables written by the new binding (i.e., $DeliveryOutputSig(p_2, c)$). The preceding theorem is applied to each case to determine whether the proposed reconfiguration is considered to be safe:

1. Write to z. This reconfiguration is not safe, since it changes the dataflow dependency graph in Figure 2: p_3 would read the value written to z by p_2, not the value written by p_1 as before (Figure 3).
2. Write to y. This reconfiguration is also not safe

Figure 3 Write to *z* reconfiguration

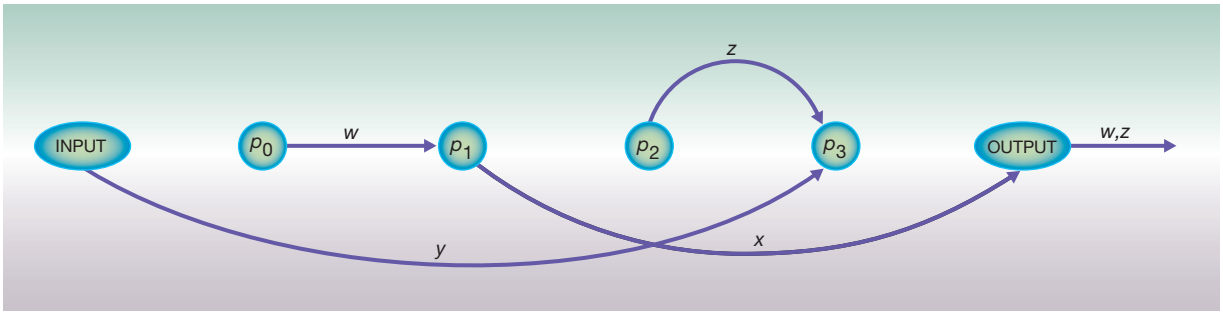
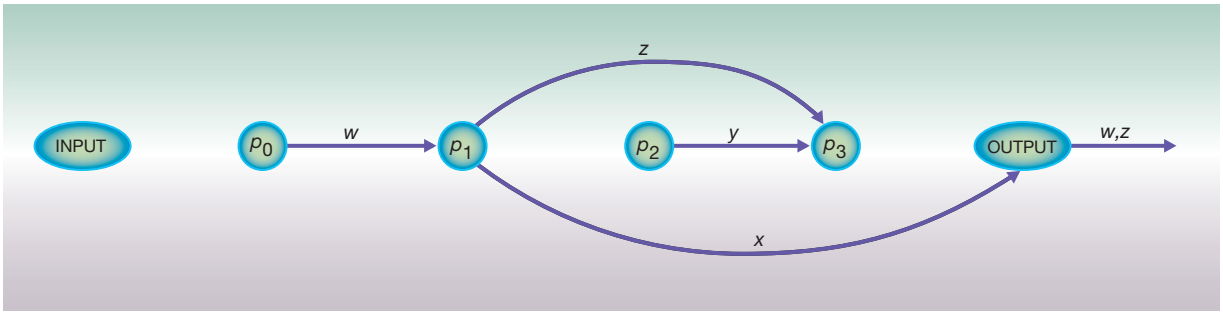


Figure 4 Write to *y* reconfiguration



for similar reasons: p_3 no longer reads the value of y established before P executes (Figure 4).

3. Write to w. This reconfiguration is safe, since no operation within P after p_2 reads from w. Since w is an intermediate variable (not the final output of P), its value will be restored when P completes; thus, the side effect of the new binding $p_2 \rightarrow c$ will be masked (Figure 5).
4. Write to v. This reconfiguration is also safe, since no operation within P after p_2 reads from v (Figure 6).

The cases involving writing w and writing v differ in one important respect: in the latter case, variable v was not included in the set of intermediate thread variables used by P, since no operation in P was known to produce v prior to the reconfiguration. Thus, there are two possible scenarios to consider:

- Sequence P was pushed onto the operation stack of the thread prior to the reconfiguration, in which case the frame stack of the thread does not contain the correct value of v to restore when P completes. Because sequence P was pushed onto the stack with the intent of being executed in an earlier configuration not containing the binding $p_2 \rightarrow c$, it is appropriate to suppress the $p_2 \rightarrow c$ delivery, thus preventing variable v from being overwritten as a side effect of the reconfiguration.
- Sequence P was pushed onto the operation stack of the thread after the reconfiguration, in which case v is known to be an intermediate value produced by P. The stack frame, therefore, contains the correct value of v to restore when P completes.

Extrasequence dependencies. The additional binding $p_2 \rightarrow c$ from Figure 2 can affect only operations that follow sequence P in the execution of the thread if $p_2 \rightarrow c$ writes to variables in the output signature of P. This is a necessary but not sufficient condition for the reconfiguration to be considered safe, since $p_2 \rightarrow c$ can write to an output variable of P as long as no other operation that follows P depends upon the overwritten value (again, the notion of preserving the dataflow dependency relationships that existed before reconfiguration).

Figure 5 Write to *w* reconfiguration

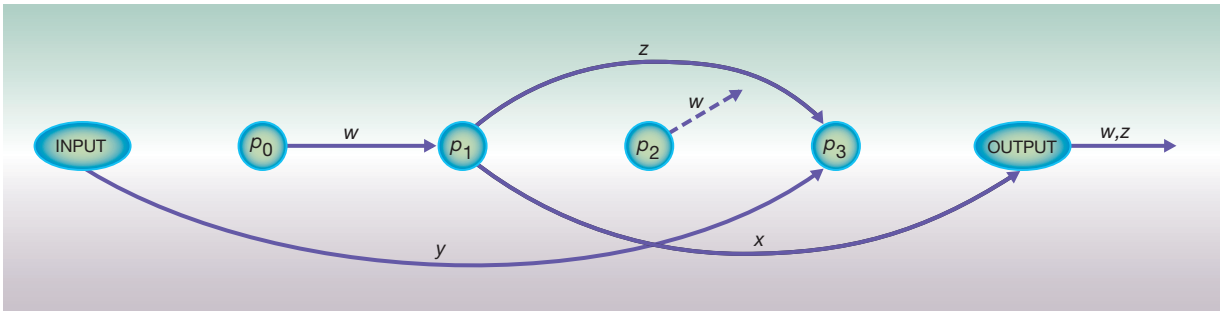


Figure 6 Write to *v* reconfiguration

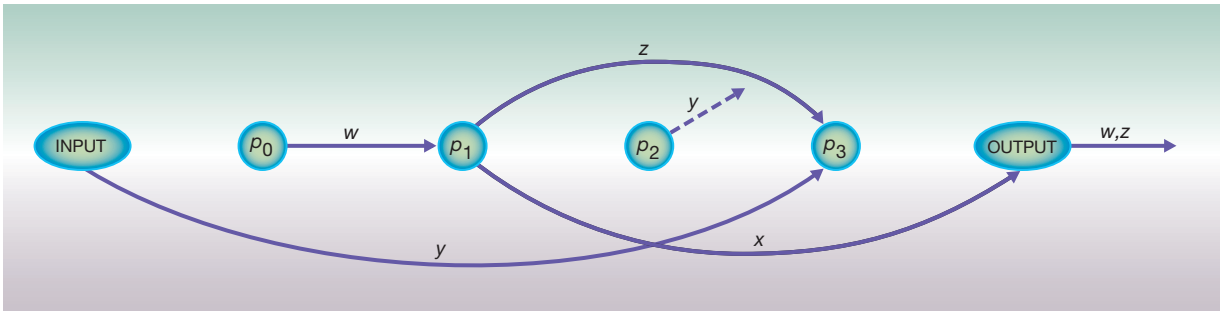


Figure 7 presents an example in which operation p executes as part of a sequence P pushed by an operation $q_j \in Q$. When sequence P completes, operations that follow q_j in sequence Q will begin executing (q_{j+1} in the figure). If p is the last operation within P to write to thread variable v, and v is in the output signature for sequence P, then the value written to v by operation p will propagate to operations that follow q in sequence Q.

Lemma 1 (Write Propagation Outside of Sequence): *Given an operation sequence P and an operation* $p \in P$, *the value written to variable* v *by* p *will propagate to operations that follow P if and only if no other operations after* p *in P overwrite variable* v. *This condition is defined by the following predicate:*

$$WritePropagation(v, p, P) \equiv v \in (ExecOutputSig(p) \cap SeqOutputSig(P)) \wedge \forall p' >_P p : v \notin ExecOutputSig(p')$$

If operation $q \in Q$ pushes operation sequence P onto the operation stack of the thread, then q is called the *parent operation* of sequence P. In general, an operation sequence can have several parents, given by the set $parents(P)$ as follows:

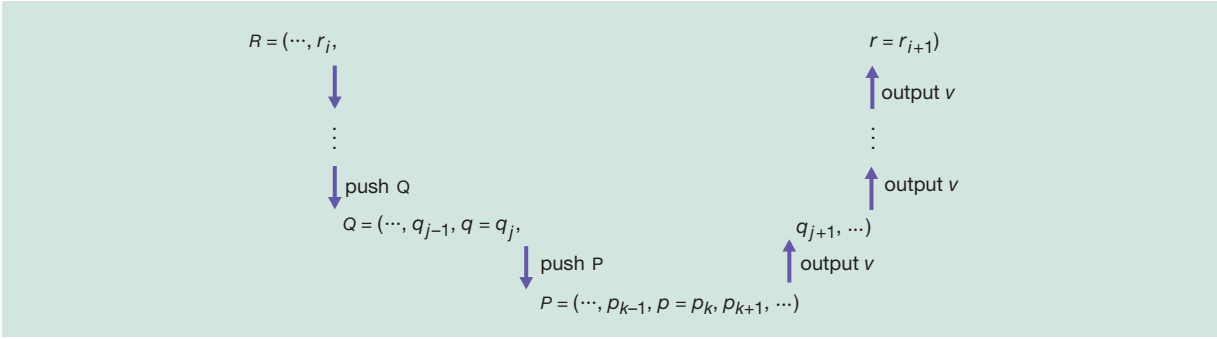
$$parents(P) \equiv \{p \in \mathcal{P} : \exists c \in \mathcal{C} : P \in c.\text{pushOps} \wedge c = BindCode(p)\}.$$

The concept of parent operations can be used to establish a "leads to" relationship between two operations and between an operation and a sequence:

- $q \rightsquigarrow P \Leftrightarrow q \in parents(P)$
- $q \rightsquigarrow p \Leftrightarrow p \in P \wedge q \in parents(P)$

Finally, $p \overset{*}{\rightsquigarrow} q$ denotes a transitive chain of "leads to" relationships, meaning that the execution of operation p brings about the execution of operation q in either one step (i.e., $p \rightsquigarrow q \Rightarrow p \overset{*}{\rightsquigarrow} q$) or several steps (i.e., $p \overset{*}{\rightsquigarrow} q \wedge q \overset{*}{\rightsquigarrow} r \Rightarrow p \overset{*}{\rightsquigarrow} r$). The "leads

Figure 7 Output variable v propagates after being written by operation p



to" relationship is needed to determine the extent to which a reconfiguration affects other operations. If $q \overset{*}{\rightsquigarrow} p$, then changes to the binding of p can propagate to operations that follow q as expressed in Lemma 2, presented below. The dataflow dependencies for each operation that leads to p must be checked in order to determine whether a reconfiguration is safe.

Continuing with the example in Figure 3, let operation $r \in R$ push sequence Q onto the operation stack of the thread when r executes; thus, $r \overset{*}{\rightsquigarrow} p$. The value in variable v that exists after operation q executes (q being the operation that pushed sequence P) will propagate to operations that follow r in sequence R if and only if the remaining operations in Q do not overwrite v and v is part of the output signature of sequence Q. This propagation continues up the $\overset{*}{\rightsquigarrow}$ chain as long as these two conditions hold at every step; as soon as one of the conditions is violated, checking can stop, since the value in v will not propagate further.[7]

Lemma 2 (Generalized Write Propagation): *Given an operation sequence P for which $r \overset{*}{\rightsquigarrow} P$, the value written to variable v by operation $p \in P$ propagates to operations that follow r if and only if the following condition holds:*

$$WritePropagation^*(v, p, P, r)$$
$$\equiv \begin{cases} WritePropagation(v, p, P), & for\ r \rightsquigarrow p \\ \exists\, q \in parents(P) : r \overset{*}{\rightsquigarrow} q \rightsquigarrow p \wedge q \in Q \Rightarrow \\ \quad WritePropagation(v, p, P) \wedge & otherwise \\ \quad WritePropagation^*(v, q, Q, r), \end{cases}$$

Proof rationale. The *WritePropagation** predicate is intended to be a generalization of the *WritePropagation* predicate introduced in Lemma 1. When r is an immediate parent of p (i.e., $r \rightsquigarrow p$), Lemma 1 can be directly applied.

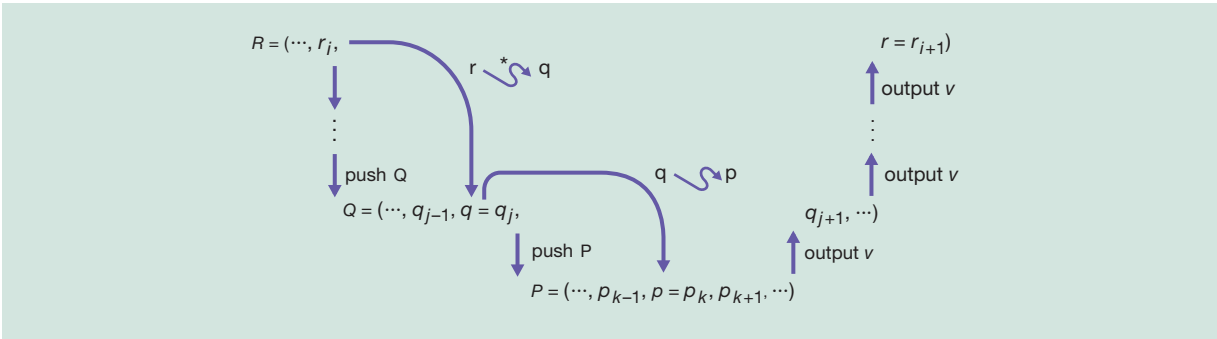
If $r \overset{*}{\rightsquigarrow} p$ forms a multistep chain (see Figure 8), then *WritePropagation** is recursively defined. The intuition is that the write to variable v performed by operation p will propagate back to operation r if there is at least one path of propagation, namely a path through operation q in which q is the parent operation of p. The write propagates from p to r if the write propagates outside sequence P, denoted by $WritePropagation(v, p, P)$, and if the write propagates from q to r, denoted by $WritePropagation^*(v, q, Q, r)$.

If the value in v propagates back to sequence R in Figure 7, then a reconfiguration that overwrites v can be considered safe as long as future operations in R do not read the overwritten value in v. This is the essence of the following theorem: a reconfiguration is considered safe only if either (1) the effects of the new binding do not propagate to sequence R (where $r \overset{*}{\rightsquigarrow} p$ and $r \in R$) or (2) the effects of the new binding propagate to sequence R, but no operations within R read from the overwritten variable.

Theorem 2 (Extrasequence Safety): *Given an operation sequence $P \in \mathcal{P}^*$ and an operation $p \in P$, the binding $p \rightarrow c$ can be safely added to the system only if any variables written by $p \rightarrow c$ do not overwrite the values in the variables expected by later operations that follow P. Formally, a safe reconfiguration implies:*

$$\forall R \in \bigcup_{c \in \mathcal{C}} c.\text{pushOps} : \forall r \in R : r \overset{*}{\rightsquigarrow} p$$
$$\Rightarrow (\forall v \in \mathcal{V} : \neg WritePropagation^*(v, p, P, r)$$

Figure 8 Value written by operation p propagates back to sequence R through intermediate sequence Q



$$\vee\ (\forall r' \succ_R r\ :\ v \in ExecInputSig(r')$$
$$\Rightarrow \exists\ r'' \in R\ :\ r \prec_R r'' \prec_R r'$$
$$\wedge\ v \in ExecOutputSig(r'')))$$

Proof rationale. All operations that lead to p must be examined to determine whether the reconfiguration is safe. Only those operations r that exist in operation sequences pushed by code blocks are considered ($r \in R$, where $R \in \cup_{c \in \mathcal{C}} c.\text{pushOps}$). For each of these operations, one of the following two conditions must hold for each thread variable for the reconfiguration to be considered safe:

1. The value written by p from sequence P must not propagate back to those operations that follow r in sequence R ($\neg WritePropagation^*(v, p, P, r)$). If the value does not propagate, then operations that follow r will not be affected by the reconfiguration.
2. If the value propagates to sequence R, then operations that follow r in R are checked to see whether they read from the variable written by p ($v \in ExecInputSig(r')$, where $r' \succ_R r$). If such an operation r' is found, then the reconfiguration can be considered safe only if there is another operation between r and r' that overwrites the value in v (i.e., r' does not read the value in v established by operation p).

Theorem 3 (Safe Reconfiguration Criteria): *Adding a new binding $p \rightarrow c$ to the system is considered safe if and only if the new binding satisfies both the Intrasequence Safety and Extrasequence Safety properties.*

Impact on element state. The previous subsection established the conditions under which a reconfiguration can be considered safe from the standpoint of thread state. In addition to thread state, the system also contains state variables found in the elements. Operations in the threads can manipulate state variables as they execute, so changing the bindings of operations to elements can impact the changes that are brought about in the element state.

Recall from the subsection on concurrency that threads lock an element before an operation delivery to ensure mutually exclusive access to the element. The element is unlocked after the delivery, permitting other threads to operate on the element. For example, the execution of an operation sequence $P = \langle p_1, p_2, p_3 \rangle$ with operation-to-element bindings[8] $p_1 \rightarrow e_1$, $p_2 \rightarrow e_2$, and $p_3 \rightarrow e_1$ is shown in Figure 9A. State changes to the elements are atomic only with respect to a single operation delivery. Another thread, for example, can deliver an operation to element e_1 in Figure 9A between operations p_1 and p_3.

Figure 9B shows how multidelivery locks can be used to extend atomicity across several operation deliveries. In this case, the lock action indicates that the lock for each element is not released after delivery until the unlock is reached, at which point all the element locks in the block are released.[9,10] With multidelivery locks, dataflow dependencies among elements can be established within each lock/unlock block. Outside these blocks, the element dataflow dependencies cannot be determined, since other threads can overwrite the element state between operation deliveries. Once the dataflow dependencies are established within the lock/unlock blocks, the analysis for determining a safe reconfiguration is similar to the procedure outlined earlier in the subsec-

Figure 9 (A) Default locking for each operation delivery; (B) element locks held across several deliveries

(A) [**lock** e_1], p_1, [**unlock** e_1], [**lock** e_2], p_2, [**unlock** e_2], [**lock** e_1], p_3, [**unlock** e_1]

(B) [**lock**], p_1, p_2, p_3, [**unlock**]

tion, "Impact on thread state," but space considerations preclude a detailed examination of the criteria.

Reconfigurations in practice

The safe reconfiguration criteria presented in the previous section require only static information for input (namely, the current and proposed configurations expressed as *BindCode* functions). Thus, these checks can be performed off line while the system executes. This section briefly describes examples in which reconfiguration is useful and how the safe reconfiguration criteria can be employed. It then describes a reconfigurable software-implemented fault-tolerant environment developed using these ideas.

Example applications of reconfigurability. The following are three types of applications in which the model is useful in determining the safety of a proposed reconfiguration:

1. Different execution phases for long-running applications. Some long-running applications require functionality that varies according to their phase of execution. A spacecraft sent to explore one of the outer planets in the solar system, for example, spends most of its time traveling to reach the target planet. While in this cruise mode, power must be conserved, and the demands on the system are few. When the spacecraft reaches the planet, however, it must perform several tasks, such as collecting data, taking pictures, navigating a fly-by of the planet, or controlling its own descent and landing. These phase-specific code blocks can be activated only when necessary through the reconfiguration concepts outlined in this paper. Since the number of phases and the transitions between phases is known at design time, the systems engineer can apply the safe reconfiguration criteria during the development cycle to verify that the intended transitions are safe.
2. On-line software upgrades. Since software upgrades involve changing the code that applications execute, on-line software upgrades can be viewed as reconfigurations to the system. Before the upgrades are made, the safe reconfiguration criteria can be used to ensure that the proposed upgrade is compatible with the existing configuration of the software.
3. Adaptivity. Some application domains require that the software adapt to changing conditions in the environment of the application. Middleware that provides fault tolerance to distributed applications, for example, may need to adjust the level of service it provides to the application depending upon the observed error behavior. Software structured around the system model presented in this paper facilitates this adaptation.

   Reconfiguration can be used to transform the computation of the application to take advantage of various mechanisms of fault tolerance (e.g., incremental checkpointing of element state,[11] alternate implementations of an element that employ design diversity to mitigate the effects of software bugs, and backup elements that store redundant copies of the data). The safe reconfiguration criteria can be used to show that the additional fault tolerance mechanisms do not disrupt the normal computation performed by the application.

Reconfigurable SIFT environment. We have developed a software-implemented fault-tolerant (SIFT) environment by employing the proposed formal system model. The SIFT environment consists of ARMOR processes, which provide error detection and recovery services to themselves and to user applications.[2,12] Because ARMOR processes are designed around the system model presented in this paper, the SIFT environment can be customized—even during run

time—to the particular dependability needs of the application.

The ARMOR-based SIFT environment was used for managing parallel scientific applications executing on a computing test bed at the Jet Propulsion Laboratory.[13] Extensive fault injection testing revealed that it is essential for the SIFT environment to be protected against errors in order to provide adequate fault-tolerance services to the application. The reconfigurability concepts introduced in this paper were applied to incorporate fault tolerance into the ARMOR processes as follows:

1. Microcheckpointing[11] was transparently added to the ARMOR processes to protect the state of the SIFT environment. The partitioning of the system state into elements permitted incremental checkpoints to be taken on an element-by-element basis. The microcheckpointing algorithm exploited the fact that state changes brought about by an operation delivery were confined to a single element and thread, thus making transparent checkpointing possible.
2. Assertion checks were added to strengthen error detection. These assertion checks were inserted during run time by dynamically changing the bindings of selected operations to pass through the assertion before being delivered to the original element. This construct was particularly useful in implementing range or sanity checks on inputs.

The safe reconfiguration criteria were applied in both of these cases to show that the added fault tolerance mechanisms did not disturb the existing functionality of the SIFT environment, which included running the scientific applications, recovering from application failures, recovering from node failures, and monitoring resource usage.

Conclusion and related work

This paper has presented a model that captures the structure (defined by elements) and run-time behavior (defined by operations) of a system. An executing code block can bring about state changes only within a single element and thread. The extent of these state changes are represented in a signature for the code block, and the collective set of signatures in the system can be used to analyze the dependencies that exist among operations and among elements.

By indirectly invoking code blocks through operations, the behavior of the system can be reconfigured by changing the operation-to-code block binding. Indirection can be achieved statically through designs such as the Polylith software bus,[14] in which components are interconnected through a mediator. The bindings of operations to code blocks is somewhat similar to a publish-and-subscribe event subsystem, except that operations are not asynchronous. Operations execute sequentially and, therefore, more closely resemble instructions that execute in a virtual machine architecture described by the proposed system model.

Architectural description languages (ADLs) are popular in describing the structure of a system.[15–17] ADLs model the system at a conceptual level, using port-based connections between components to define the system structure. Some incorporate semantics that describe the behavior of the components and connectors, including constraints in their usage. Our model, in contrast, is rooted in the implementation of the system and is a bottom-up approach in which the programmer describes the behavior of code blocks through the use of per-block signatures. These signatures are processed in an automated fashion to construct dependency relationships between the system components. Additional code blocks can be designed without the need to formally incorporate them into a larger architectural description model—only the signatures for the new code blocks are required to apply the safe reconfiguration criteria presented earlier.

Nevertheless, several ADLs express dynamic reconfigurations at the architectural level. Darwin, for example, addresses the problem from a structural perspective by allowing components to be instantiated during run time,[18] but reconfigurations only occur while the system is quiescent to preserve consistency. Wright permits the architectural topology of the system to change during run time in response to special control events, which are distinguished from the usual communication events that drive its component behavior.[19] It has also been suggested that architectural styles and models can be incorporated into the adaptation framework of self-repairing systems as first-class entities.[20] The architectural styles are used to determine what aspects of the system should be monitored and how to reconfigure the system within predetermined constraints.

Shrivastava and Wheater describe a reconfigurable workflow model that provides run-time support for interconnecting the I/O of executing tasks.[21] Although the work outlines the infrastructure that supports re-

configuration, there is no mention of how to judge whether a proposed reconfiguration can be considered safe given the current configuration of the system. Reconfiguration compatibility issues have been addressed with respect to real-time control applications in Feiler and Li,[22] but this analysis requires knowledge of the semantics and permissible behavior of the application.

Acknowledgments

This work was supported in part by NASA Fellowship NGT5-50228 for Keith Whisnant, NSF grants CCR 00-86096 ITR and CCR 99-02026, grant ARMY WMH 0993, and a grant from Motorola. Special thanks to Fran Baker for reviewing early drafts of this paper.

Cited references and notes

1. It may be that the user is reconfiguring the system with the explicit intent of changing the dataflow dependencies; thus, the off-line tests are used only as warnings of possible incompatibility.
2. 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**, No. 6, 560–579 (June 1999).
3. Where appropriate, the *name* of a variable $v \in \mathcal{V}$ (v.name) is distinguished from the *value* of the variable (v.val).
4. If a code block both writes to thread variables and pushes an operation sequence onto the operation stack of the thread while operation p is delivered, then the operations in the pushed sequence begin executing only after p completely executes. The thread variables written by p, therefore, are visible to the operation sequence pushed by p.
5. When the boundaries of an atomic state change must extend beyond a single operation delivery, multidelivery locks can be used as described in a later section.
6. Dataflow dependencies can be formally defined between an operation q and an earlier operation p with respect to a particular thread variable v.
7. This situation can be compared to a series of nested function calls: The return value for the innermost function call only propagates to the topmost level if it is returned unmodified by each function call in the chain.
8. Since each code block exists in one and only one element, operation-to-element bindings can be derived from the *BindCode* function.
9. Details of these multidelivery locks (such as deadlock detection and avoidance[10]) are beyond the scope of this paper, but the issues are similar to those found in traditional concurrency control and parallel programming.
10. C.-S. Shih and J. Stankovic, *Survey of Deadlock Detection in Distributed Concurrent Programming Environments and Its Application to Real-Time Systems and Ada*, Technical Report UM-CS-1990-069, University of Massachusetts, Amherst, MA (August 1990).
11. K. Whisnant, Z. Kalbarczyk, and R. K. Iyer, "Micro-checkpointing: Checkpointing for Multithreaded Applications," *Proceedings of the 6th IEEE International On-Line Testing Workshop* (July 2000).
12. 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**, No. 2, 203–224 (March/April 2000).
13. K. Whisnant, R. K. Iyer, P. Jones, R. Some, and D. Rennels, "An Experimental Evaluation of the REE SIFT Environment for Spaceborne Applications," *Proceedings of the 2002 International Dependable Systems and Networks* (June 2002), pp. 585–595.
14. J. Purtilo, "The Polylith Software Bus," *ACM Transactions on Programming Languages and Systems* **16**, No. 1, 151–174 (January 1994).
15. D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture Using Rapide," *IEEE Transactions on Software Engineering* **21**, No. 4, 336–355 (April 1995).
16. N. Medvidovic and R. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering* **26**, No. 1, 70–93 (January 2000).
17. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," *IEEE Transactions on Software Engineering* **21**, No. 4, 314–335 (April 1995).
18. J. Kramer and J. Magee, "Analysing Dynamic Change in Software Architectures: A Case Study," *Proceedings of the Fourth International Conference on Configurable Distributed Systems* (1998), pp. 91–100.
19. R. Allen, R. Douence, and D. Garlan, "Specifying and Analyzing Dynamic Software Architectures," *Proceedings of the Conference on Fundamental Approaches to Software Engineering*, *Lecture Notes in Computer Science*, Vol. 1382 (April 1998).
20. S.-W. Cheng, D. Garlan, B. Schmerl, J. P. Sousa, B. Spitznagel, and P. Steenkiste, "Using Architectural Style as a Basis for System Self-Repair," *Proceedings of the Working IEEE/IFIP Conference on Software Architecture* (2002), pp. 45–59.
21. S. Shrivastava and S. Wheater, "Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications," *Proceedings of the 4th International Conference on Configurable Distributed Systems* (May 1998), pp. 10–17.
22. P. Feiler and J. Li, "Consistency in Dynamic Reconfiguration," *Proceedings of the 4th International Conference on Configurable Distributed Systems* (May 1998), pp. 189–196.

Accepted for publication September 16, 2002.

Keith Whisnant *Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1308 W. Main Street, Urbana, Illinois 61801 (electronic mail: kwhisnan@crhc.uiuc.edu).* Mr. Whisnant is currently a Ph.D. student at the University of Illinois. He holds a B.S. and an M.S. degree in computer engineering, also from the University of Illinois. While in graduate school, he has been the lead architect and system designer for the Chameleon ARMORs project, which uses a reconfigurable software infrastructure to provide a wide variety of fault-tolerant services to user applications. He has worked with the Jet Propulsion Laboratory in applying the ARMOR concepts to a software-implemented fault-tolerant environment for protecting parallel scientific applications that execute in space.

Zbigniew T. Kalbarczyk *Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1308 W. Main Street, Urbana, Illinois 61801 (electronic mail: kalbar@*

crhc.uiuc.edu). Dr. Kalbarczyk is currently Principal Research Scientist at the Center for Reliable and High-Performance Computing in the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign. He holds a Ph.D. in computer science. After receiving his doctorate, he worked as Assistant Professor in the Laboratory for Dependable Computing at Chalmers University of Technology in Gothenburg, Sweden. Currently he is one of the leading researchers on the project to explore and develop approaches and techniques for providing an adaptive software infrastructure that allows different levels of availability requirements to be supported in a network environment. Dr. Kalbarczyk's research also involves developing automated fault/error injection techniques for validation and benchmarking of dependable and secure computing systems. He has served as a program co-chair of the International Performance and Dependability Symposium (IPDS), a track of the Conference on Dependable Systems and Networks (DSN 2002), and is regularly invited to work on the program committees of major conferences on design of fault-tolerant systems.

Ravishankar K. Iyer *Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1308 W. Main Street, Urbana, Illinois 61801 (electronic mail: iyer@ crhc.uiuc.edu).* Professor Iyer is Director of the Coordinated Science Laboratory (CSL) at the University of Illinois at Urbana-Champaign, where he is George and Ann Fisher Distinguished Professor of Engineering. He holds appointments in the Department of Electrical and Computer Engineering and the Department of Computer Science. He is Co-Director of the Center for Reliable and High-Performance Computing at CSL. His research interests are in the area of reliable networked systems. He currently leads the Chameleon ARMORs project at the University of Illinois, which is developing adaptive architectures for supporting a wide range of dependability and security requirements in heterogeneous networked environments. Professor Iyer is a Fellow of the IEEE and the ACM, and an Associate Fellow of the American Institute for Aeronautics and Astronautics (AIAA). He has received several awards, including the Humboldt Foundation Senior Distinguished Scientist Award for excellence in research and teaching, the AIAA Information Systems Award and Medal for "fundamental and pioneering contributions towards the design, evaluation, and validation of dependable aerospace computing systems," and the IEEE Emanuel R. Piore Award "for fundamental contributions to measurement, evaluation, and design of reliable computing systems."