A Functional and Attribute Based Computational Model for Fault-Tolerant Software

Masato Suzuki, Takuya Katayama,[1] and Richard D. Schlichting

TR 93-8

Abstract

Programs constructed using techniques that allow software or operational faults to be tolerated are typically written using an imperative computational model. Here, an alternative is described in which such programs are written using a functional and attribute based model called FTAG. This approach offers several advantages, including a declarative style, separation of semantic and syntactic definitions, and the simplicity of a functional foundation. While important for any type of programming, these advantages are especially pronounced for writing fault-tolerant programs that involve the use of state rollback, including the recovery block technique for software faults and checkpointing for operational faults. A pure value reference model is described in which *redoing* is introduced as a fundamental operation. A formal description of the model is also given, together with an outline of how this model can be implemented in a computer system containing multiple processors. Several rollback-oriented examples are used to illustrate the model.

March 8, 1993

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

[1]The first two authors are affiliated with the School of Information Science, Japan Adv. Inst. of Sci. and Tech., 15 Asahidai, Tatsunokuchi, Ishikawa, 923-12, Japan.

1 Introduction

Fault-tolerant software is software that is constructed to continue providing service despite the existence of software faults (i.e., program bugs) and/or operational faults (i.e., faults in the underlying computing platform.) Over the years, a variety of techniques, mechanisms, and structuring paradigms have been developed for building software of this type. These include such things as *recovery blocks* [Ran75] and *N-version programming* [Avi85] for dealing with software faults, and *checkpointing* [BHG87], *atomic actions* [Lis85], and the *replicated state machine approach* [Sch90] for dealing with operational faults. All of these simplify the problems associated with faults by providing the programmer with higher-level models or abstractions.

Despite the inherent differences in these approaches, one common thread is that they all have typically been conceived and expressed using an imperative computational model. In this paper, we describe FTAG, an alternative in which fault-tolerant software is written using a functional and attribute based model. The model is derived from an existing collection of models that use an attribute grammar formalism [Knu68] for such diverse purposes as functional programming [Kat81, SK88], object-oriented programming [SK90a], and modeling of the software development process [Kat89]. This approach offers several advantages, including a declarative style, separation of semantic and syntactic definitions, and the simplicity of a functional foundation. While important for any type of programming, these advantages are especially pronounced for writing fault-tolerant programs that involve the use of state rollback, such as those that use recovery blocks or checkpointing. The experience modeling the software process cited above is especially relevant in this regard, since many of the characteristics of higher level software development—for example, the possibility of incomplete or incorrect execution of a development step due to human error—have natural analogues in fault-tolerant software.

This paper is organized as follows. In Section 2, a pure value reference model for FTAG is described in which *redoing* [SK90b] is introduced as a fundamental operation; this operation, together with the functional characteristic of the model, provide the fundamental requirements for performing state rollback and recovery. This is followed in Section 3 by a description of how this model can be used for fault-tolerance; several rollback-oriented examples are given to illustrate the process. Section 4 then gives a formal description of the FTAG model. An outline of how this model can be realized as a practical programming system is given in Section 5; this includes describing a system model involving multiple processors and an object base, and giving a mapping from the reference model to the system model. Finally, Section 6 contains conclusions and directions for future work.

2 Value Reference Model

In this section, we introduce the FTAG computational model. The model is based on the HFP (Hierarchical and Functional Process) model [Kat81], which is in turn derived from attribute grammars [Knu68]. The basics of the model are outlined first, followed by a description of redoing.

2.1 The FTAG Computational Model

In FTAG, every computation consists of a collection of pure mathematical functions called $modules$. Each module has multiple inputs and outputs. A module M with inputs $x_1, \ldots, x_n$ and outputs $y_1, \ldots, y_m$ is denoted by

$$M(x_1, \ldots, x_n \mid y_1, \ldots, y_m)$$

We call $x_1, \ldots, x_n$,$y_1, \ldots, y_m$ the $attributes$ of M, where $x_1, \ldots, x_n$ are $inherited\ attributes$ and $y_1, \ldots, y_m$ are $synthesized\ attributes$.

When M is simple enough to be performed directly, we call it a $primitive\ module$ and denote it by

$$M(x_1, \ldots, x_n \mid y_1, \ldots, y_m) \Rightarrow return\ \ \mathbf{where}\ \ E$$

where E is a collection of equations by which $y_1, \ldots, y_m$ are calculated from $x_1, \ldots, x_n$. Otherwise, M is decomposed into submodules. To do this, the way in which M is decomposed into submodules $M_1, .., M_k$, and the relationship among inputs and outputs of M and M_i are specified as follows:

$$M(x_1, \ldots, x_n \mid y_1, \ldots, y_m) \Rightarrow M_1 \ldots M_k\ \ \mathbf{where}\ \ E$$

Here, E is a collection of equations that denotes the relationship among the inherited attributes of M_i and the synthesized attributes of M. This explicit definition of E is sometimes omitted, in which case the following convention is used: when an output y of the function M_i is transferred to M_j as one of its inputs, say x, we omit the definition $x = y$ and simply put y for x in M_j. A pair of $M_1 \ldots M_k$ and E are called a $decomposition$ and denoted by D hereafter.

Sometimes a module decomposition is specified with a condition C controlling when this decomposition is to be applied, leading to the following general form.

$$\begin{array}{l} M(x_1, \ldots, x_n | y_1, \ldots, y_m) \Rightarrow \\ \quad \mathbf{case} \\ \quad\quad \begin{array}{ll} C_1 & \Rightarrow D_1 \\ \quad\vdots & \\ C_n & \Rightarrow D_n \\ \mathbf{otherwise} & \Rightarrow D_{def} \end{array} \\ \quad \mathbf{end} \end{array}$$

The conditions $C_1, \ldots, C_n$ are tested sequentially with the decomposition D_i being applied when C_i is satisfied. If none of $C_1, \ldots, C_n$ are satisfied, the default decomposition D_{def} is selected and performed.

Execution of an FTAG program is performed by successively applying the above module decomposition process until every module is decomposed into primitive modules. The synthesized attributes are then calculated based on the given equations and returned. Hence, the resulting execution takes the form of a $computation\ tree$ in which inherited attributes flow down the tree and synthesized attributes up.

Figure 1 shows how quicksort might be expressed using FTAG, with Figure 2 being the corresponding computation tree. In Figure 1, $\hat{}seq$ is used to denote the first value of the sequence seq, $\tilde{}seq$ the sequence except the first value, and $seq.1 + seq.2$ the concatenation of sequences.

The order in which modules are decomposed is determined solely by attribute dependencies among submodules, making this model highly amenable to parallel execution. As an example, consider the **otherwise** clause in the *qsort* module of Figure 1. There are only two dependencies among the submodules labeled (1),(2),(3) and (4): between (1) and (3) due to $seq.less$, and between (2) and (4) due to $seq.grtr$. Thus, in this case, it is guaranteed only that (3) is executed after (1), and (4) after (2); the execution order of (1) and (2), and (3) and (4) is indeterminate.

Besides this implicit ordering, FTAG has features for explicitly ordering module decomposition. For example, we can enforce the execution of M_1 before M_2 by using the sequencing operator ';' and the grouping construct $\{\}$, as in $\{M_1; M_2\}$. Thus, to force (2) to execute after (1) in Figure 1, we could change the **otherwise** clause as follows:

```
type
  seq = sequence of x
  x = anytype

qsort(seq.in|seq.out) ⇒
  case
    when (seq.in ==<>) or
       (˜seq.in ==<>) ⇒
      return
    where
      seq.out = seq.in
    otherwise ⇒
      less_half(ˆseq.in, ˜seq.in|seq.less)      (1)
      grtr_half(ˆseq.in, ˜seq.in|seq.grtr)      (2)
      qsort(seq.less|seq.lessout)               (3)
      qsort(seq.grtr|seq.grtrout)               (4)
    where
      seq.out = seq.lessout
      + < ˆseq.in > +seq.grtrout
  end

less_half(x.ref, seq.in|seq.out) ⇒
  case
    when seq.in ==<>⇒
      return
    where
      seq.out = seq.in
    when ˆseq.in >= x.ref ⇒
      less_half(x.ref, ˜seq.in|seq.out)
    where
      (* No attribute equations *)
    when ˆseq.in < x.ref ⇒
      less_half(x.ref, ˜seq.in|seq.subout)
    where
      seq.out =< ˆseq.in > +seq.subout
    otherwise ⇒
      return
    where
      seq.out = seq.in
  end

grtr_half(x.ref, seq.in|seq.out) ⇒
  case
    when seq.in ==<>⇒
      return
    where
      seq.out = seq.in
    when ˆseq.in < x.ref ⇒
      grtr_half(x.ref, ˜seq.in|seq.out)
    where
      (* No attribute equations *)
    when ˆseq.in >= x.ref ⇒
      grtr_half(x.ref, ˜seq.in|seq.subout)
    where
      seq.out =< ˆseq.in > +seq.subout
    otherwise ⇒
      return
    where
      seq.out = seq.in
  end
```

Figure 1: Quicksort in FTAG

```
otherwise ⇒
  {less_half(ˆseq.in, ˜seq.in|seq.less);
    grtr_half(ˆseq.in, ˜seq.in|seq.grtr)}   (1; 2)
  qsort(seq.less|seq.lessout)               (3)
  qsort(seq.grtr|seq.grtrout)               (4)
```

Submodules (1) and (2) are now combined into (1;2) and executed sequentially, with the combination being treated as a single module with input $seq.in$ and outputs $seq.less$ and $seq.grtr$. The execution order of (3) and (4) is still indeterminate due to lack of dependencies.

There are several advantages to the functional approach used in FTAG as opposed to traditional imperative styles. One is that it is static and declarative. This makes reading and understanding the description easier, and simplifies incremental creation and evolution of the program. The separation of syntactic and semantic definitions also contributes to the readability of the program. Another is that a module decomposition only represents the structure of the computation and relationships among the attributes, rather than the way the computation is performed. Finally, the program exhibits a high degree of locality; information is only passed

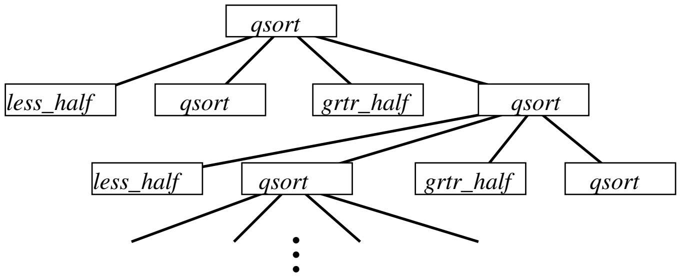

Figure 2: Quicksort Computation Tree

between functions using attributes, and only then between functions that have a parent/child relationship. As a result, it is easy to determine attribute dependencies, facilitating, for example, parallel execution as noted above.

2.2 Redoing

In this section, we introduce the notion of $redoing$, in which a portion of the computation tree is replaced by a new computation. Although redoing has many uses, here we focus on its use as a mechanism for replacing a part of the computation that has failed—that is, generated incorrect results or no results at all—with a new computation. Here, we assume that all failures of interest are manifested by incorrect attribute values that can be tested by a conditional. Such an assumption is common for software faults [Ran75], while operational faults such as crashed processors can be translated into a distinguished value $\perp$ ("bottom") that is assigned to the appropriate attribute values by the underlying system. Such a facility for operational faults is outlined in Section 5.

The simplest case of redoing is illustrated in Figure 3. Here, a module M is decomposed into M_1, M_2, and M_3, with a failure being detected at some point during the execution of M_3. Assume that an analysis determines that the failure has been introduced during the execution of M_1. Then, the whole computation starting at M_1 is discarded, and M_1 and the successive computations M_2 and M_3 are reexecuted.[1] We call this kind of special decomposition a $redoing\ decomposition$. After the new computation has completed successfully, it is regarded as a part of current active computation history. The type of redoing decomposition shown in Figure 3 would be written in a program as follows:

$$
\begin{array}{l}
M(x \mid y) \Rightarrow \\
\qquad M_1(x_1 \mid y_1)\ M_2(x_2 \mid y_2)\ M_3(x_3 \mid y_3) \\
\quad \textbf{where} \\
\qquad \vdots \\
M_3(x \mid y) \Rightarrow \\
\quad \textbf{case} \\
\qquad \begin{array}{ll} notfail(x) & \Rightarrow M_3body(x \mid y) \\ \textbf{otherwise} & \Rightarrow \textbf{redo}\ M \end{array} \\
\quad \textbf{end}
\end{array}
$$

In M_3, the condition $notfail$ tests the value of the input attributes x to verify that a failure has not occurred.

[1] In this simple case, of course, we are assuming that the failure was transient, so that executing it a second time will produce the correct value.

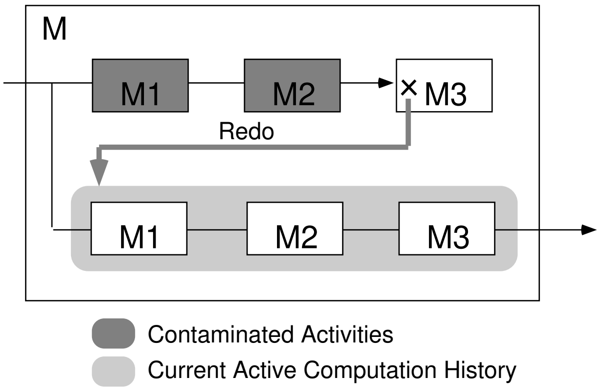

Figure 3: Redoing

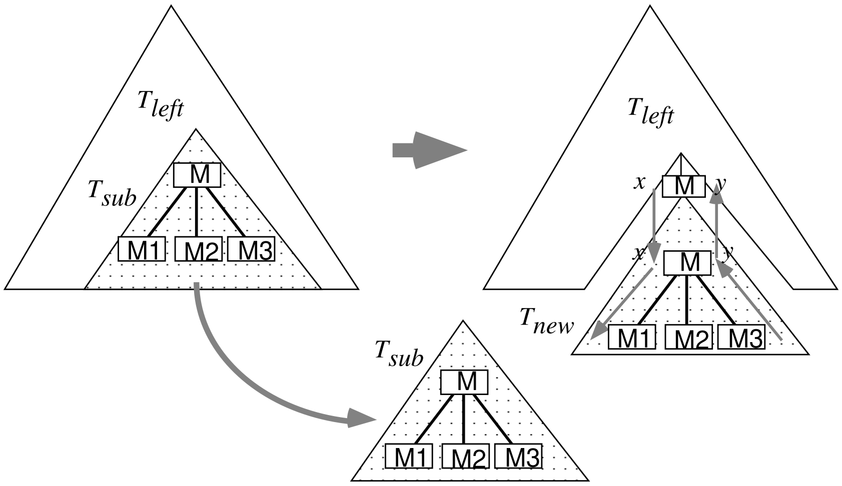

Figure 4: A Computation Tree when Redoing Occurs

If true, M_3 is decomposed into M_3body, which performs the actual function. Otherwise, M_3 is decomposed into M using the redoing operator in order to replace the previous execution of M.

Figure 4 illustrates the effect of a redoing operation on the computation tree corresponding to this example. The redoing operation starts by deleting the subtree T_{sub}, which contains improper attribute values, and produces T_{left} as the rest of the computation tree. A new tree T_{new} whose root is a new instance of the module M is then created. T_{new} is grafted to T_{left} at the appropriate place, with the inputs of M in T_{left} being passed to the new M in T_{new}. After the redoing operation has completed successfully, the results of M in T_{new} are passed to M in T_{left} as if they were the correct results of the original computation. Sometimes there are many instances of M in the computation tree. The particular one replaced by the reexecution is determined by an analysis on the computation tree. In this example, the target of redoing is the most recent execution of M, defined as the first instance on the path from M_3 to the root.

A module M can also be replaced by another module M' during redoing using the **with** clause.

$$
\begin{array}{l}
M_3(x \mid y) \Rightarrow \\
\quad \textbf{case} \\
\qquad \begin{array}{ll} notfail(x) & \Rightarrow M_3body(x \mid y) \\ \textbf{otherwise} & \Rightarrow \textbf{redo}\ M\ \textbf{with}\ M' \end{array} \\
\quad \textbf{end}
\end{array}
$$

In this case, M' is instantiated as the root of T_{new} in place of M. Of course, the number and types of the inherited and synthesized attributes of M and M' must be identical for proper grafting.

In general, redoing decompositions are useful for checking input values before the invocation of a module to guarantee that the operation is meaningful, and the output values after the invocation to guarantee that the proper operation has been performed. We can describe the combination of pre- and post-checking by a nested case-structure as follows:

$$
\begin{array}{l}
M_3(x \mid y) \Rightarrow \\
\textbf{case} \\
\quad \begin{array}{ll} \neg C_{pre}(x) & \Rightarrow \textbf{redo}\ M_{pre} \\ \textbf{otherwise} & \Rightarrow \{M_3body(x \mid y); \\ & \qquad \begin{array}{l} \textbf{case} \\ \quad \begin{array}{ll} \neg C_{post}(y) & \Rightarrow \textbf{redo}\ M_{post} \\ \textbf{otherwise} & \Rightarrow return \end{array} \\ \textbf{end}\ \} \end{array} \end{array} \\
\textbf{end}
\end{array}
$$

M_{pre} and M_{post} are modules with which we begin the recomputation when C_{pre} or C_{post} are not satisfied, respectively. In the event that input x contains errors and does not satisfy C_{pre}, M_3 is decomposed into M_{pre}. Otherwise, M_3body is executed and the output y produced. If y, in turn, does not satisfy the condition C_{post}, we assume that an error has occurred and reexecute starting at M_{post}.

3 Using FTAG for Fault-Tolerance

The functional model of FTAG together with redoing capabilities make it well-suited for many types of fault-tolerance. The advantage of the functional approach is that the computation is performed by the growth of the tree using no other global information, so that a program's entire execution status and history is available within the tree. Redoing, of course, captures the state rollback aspect used in various fault-tolerance techniques; the first step prunes the subtree containing the modules that failed, while the second does the appropriate recovery action. The state rollback is implicit and automatic since the remaining parts of the tree contain all the input values needed to redo the calculation. To illustrate these points, we now describe how redoing can be used to implement *recovery blocks*, a technique used primarily to handle software faults [Ran75], and *checkpointing*, a technique used to recover from operational faults [BHG87].

In the recovery block method, multiple implementations $M_1, \ldots, M_k$ are prepared for a module M. Execution of the multiple versions is done serially in such a way that if an *acceptance test* following M_i fails due to a failure, the state is rolled back and the next implementation M_{i+1} is performed. Such a construct

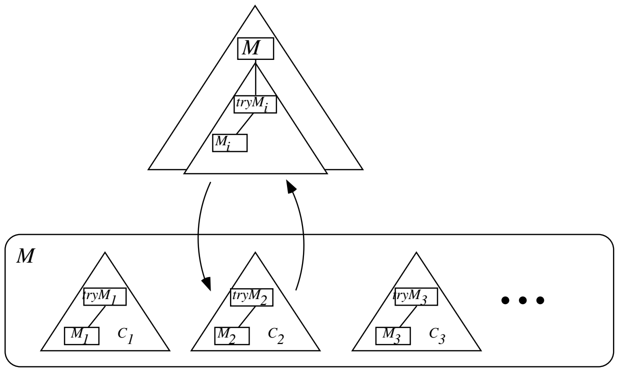

Figure 5: Computation Tree for Recovery Blocks

can be realized using redoing in FTAG as follows:

$$
\begin{array}{l}
M(x \mid y) \Rightarrow try\_M_1(x \mid y) \\
\\
try\_M_1(x \mid y) \Rightarrow \\
\quad \{M_1(x \mid y); \\
\quad\quad \textbf{case} \\
\quad\quad\quad \textbf{when } AT_1(y) \Rightarrow return \\
\quad\quad\quad \textbf{otherwise} \Rightarrow \textbf{redo } try\_M_1 \textbf{ with } try\_M_2 \\
\quad\quad \textbf{end} \quad \} \\
\\
try\_M_2(x \mid y) \Rightarrow \\
\quad \{M_2(x \mid y); \\
\quad\quad \textbf{case} \\
\quad\quad\quad \textbf{when } AT_2(y) \Rightarrow return \\
\quad\quad\quad \textbf{otherwise} \Rightarrow \textbf{redo } try\_M_2 \textbf{ with } try\_M_3 \\
\quad\quad \textbf{end} \quad \} \\
\vdots \\
try\_M_k(x \mid y) \Rightarrow \\
\quad M_k(x \mid y)
\end{array}
$$

Here, $try\_M_1, \ldots, try\_M_k$ are used to encapsulate the k different implementations of M, and $AT_1, \ldots, AT_{k-1}$ are conditionals that serve as the acceptance tests. Figure 5 shows the corresponding computation tree. The net result is that try_M_1 through try_M_k are attempted until one passes its acceptance test, with failed modules being replaced in succession using redoing decompositions. We emphasize again that no explicit state saving or restoration is needed here given the functional nature of the model. To

$$
\begin{array}{l}
\textbf{type } seq = \textbf{seqence of } x \\
sort(seq.in \mid seq.out) \Rightarrow \\
\quad \textbf{ensure} \\
\quad\quad sorted?(seq.out) \textbf{ by } quick(seq.in \mid seq.out) \\
\quad\quad sorted?(seq.out) \textbf{ by } linear(seq.in \mid seq.out) \\
\quad \textbf{end} \\
\\
sorted?(seq.out|bool) \Rightarrow \\
\quad \textbf{for } i = 1..N-1 \textbf{ do} \\
\quad\quad return \\
\quad \textbf{where} \\
\quad\quad bool = \textbf{conjunction of } seq.out[i+1] > seq.out[i] \\
\quad \textbf{end} \\
quick(seq.in|seq.out) \Rightarrow ...(*\ quick\ sort\ *) \\
linear(seq.in|seq.out) \Rightarrow ...(*\ linear\ sort\ *)
\end{array}
$$

Figure 6: Using Recovery Blocks for Sorting

improve readability, the following syntax can also be used:

$$
\begin{array}{l}
M(x \mid y) \Rightarrow \\
\quad \textbf{ensure} \\
\quad\quad AT_1(y) \ \textbf{by} \ M_1(x \mid y) \\
\quad\quad AT_2(y) \ \textbf{by} \ M_2(x \mid y) \\
\quad\quad \vdots \\
\quad\quad \textbf{otherwise} \ M_k(x \mid y) \\
\quad \textbf{end}
\end{array}
$$

As a more concrete example, consider using recovery blocks to sort numbers into ascending order, as shown Figure 6. The first implementation uses quicksort, which is faster, while the second uses linear sort, which is less complex. Suppose further that quicksort is incorrectly implemented as follows:

$$
\begin{array}{l}
grtr\_half(x.ref, seq.in|seq.out) \Rightarrow \\
\quad \textbf{case} \\
\quad\quad \textbf{when } seq.in == <> \Rightarrow \\
\quad\quad\quad return \\
\quad\quad \textbf{where} \\
\quad\quad\quad seq.out = seq.in \\
\quad\quad \textbf{when}\hat{\ } seq.in < x.ref \Rightarrow \\
\quad\quad\quad grtr\_half(x.ref, \tilde{\ } seq.in|seq.out) \\
\quad\quad \textbf{where} \\
\quad\quad\quad (*\ No\ attribute\ equations\ *) \\
\quad\quad \textbf{when}\hat{\ } seq.in > x.ref \Rightarrow (*\ Error\ !!\ *) \\
\quad\quad\quad grtr\_half(x.ref, \tilde{\ } seq.in|seq.subout) \\
\quad\quad \textbf{where} \\
\quad\quad\quad seq.out = < \hat{\ } seq.in > + seq.subout \\
\quad\quad \textbf{otherwise} \Rightarrow \\
\quad\quad\quad return \\
\quad\quad \textbf{where} \\
\quad\quad\quad seq.out = seq.in \\
\quad \textbf{end}
\end{array}
$$

The output value $seq.out$ is incorrect whenever the first element of the sequence $seq.in$ is equal to the reference value $x.ref$. In this case, the acceptance test $sorted?$ rejects the erroneous result and uses redoing to invoke the linear sort. Note that no explicit rollback mechanism is needed here: the values as they existed before sorting are automatically available where the new subtree is grafted into the original computation tree.

Checkpointing to recover from operational faults can also be implicitly implemented using the redoing mechanism, since every state of the computation is captured by a node in the computation tree. Consider the following program:

$$\begin{array}{ll} M(x \mid y) & \Rightarrow M_1(x \mid u)\ \ M_2(u \mid y) \\ M_2(x \mid u) & \Rightarrow M_{21}(x \mid v)\ \ M_{22}(v \mid u) \end{array}$$

The node in the tree corresponding to M_2 captures the status just after the execution of M_1, while M_{22} similarly captures the status just after M_{21}. Hence, M_{21} can be made a *restartable action* [Lam81] as follows by having M_{22} check its input attribute and redo M_2 should it have the value $\perp$.

$$\begin{array}{l} M_{22}(v \mid u) \Rightarrow \\ \quad \mathbf{case} \\ \qquad \mathbf{when}\ v \neq \perp \quad \Rightarrow \mathbf{redo}\ M_2 \\ \qquad \mathbf{otherwise} \qquad \Rightarrow M_{22}body(v \mid u) \\ \quad \mathbf{end} \end{array}$$

To illustrate this use of FTAG further, consider the outline of a long running scientific application shown in Figure 7. This program calculates a large table of numbers, where the cost of calculating each $V[i]$ is

$$\begin{array}{l} \mathbf{const}\ N\ = 1000000\ \ (*a\ large\ number*) \\ \mathbf{type}\ V\ \ = \mathbf{array}\ 1..N\ \mathbf{of}\ v \\ \\ calc(i \mid v) \Rightarrow ...\ \ (*an\ expensive\ calculation*) \\ \\ main(\ \mid V) \Rightarrow \\ \quad \mathbf{for}\ i = 1..N\ \mathbf{do}\ calc(i \mid v) \\ \quad \mathbf{where}\ V[i] = v \\ \quad \mathbf{end} \end{array}$$

Figure 7: An Expensive Calculation without Redoing

assumed to be non-trivial. With this organization, should an operational failure such as a crash occur during its calculation, the entire program would need to be reexecuted. Such an expensive proposition can be avoided by using the redoing capability of FTAG, as shown in Figure 8. In this case, if a failure occurs during the execution of a particular iteration, execution is rolled back only to the beginning of that iteration. This is achieved by changing $calc$ to $Calc$, which does the calculation and then checks for failure and executes a redoing decomposition if necessary. While again no explicit state saving or restoration is needed, this use of redoing does imply that the attribute values needed to reexecute $Calc$ are stored in *stable storage* to guarantee persistence across operational failures [Lam81]; in Section 5, we outline a *stable object base* abstraction that serves this purpose for FTAG. Note also that it would be possible to structure this calculation so that rollback points are established every i iterations to minimize use of this stable object base rather than every iteration as done here.

$main(\mid V) \Rightarrow$
  for $i = 1..N$ **do** $Calc(i \mid v)$
  where $V[i] = v$
  end

$Calc(i \mid v) \Rightarrow$
  $\{calc(i \mid v);$
  case
    when $v = \perp \Rightarrow$ **redo** $Calc$
    otherwise $\Rightarrow$ $return$
  end $\}$

Figure 8: An Expensive Calculation Using Redoing

4 Formal Description

In this section, we provide a formal description of the FTAG value reference model outlined in the previous sections. The following notation will be used in this description:

1. Classes and Instances: In the following definitions, every data object has a class to which it belongs. $a : A$ denotes that a data object a is a member of a class A, where a is called an instance of A.

2. Powersets: A set of all subsets of a set A is called a $powerset$ of A and is denoted by $\mathcal{P}(A)$. An element of $\mathcal{P}(A)$ is denoted by $\{ a_1, \ldots, a_k \}$, where $a_i \in A$.

3. Tuples: A tuple with components $t_1, \ldots, t_n$ is denoted by $(t_1, \ldots, t_n)$. When $t_i : T_i$, then $t = (t_1, \ldots, t_n) : T_1 \times \ldots \times T_n$.

4. Composition of Mappings: For two mappings $M_1, M_2 : A \rightarrow B$ such that $dom(M_1) \cap dom(M_2) = \phi$, the composed mapping that has all images of any elements M_1 and M_2 is denoted by $M_1 \cup M_2$. More precisely, for $a : A$, $M = M_1 \cup M_2$,

$$M(a) = \begin{cases} M_1(a) & if\ a \in dom(M_1) \\ M_2(a) & if\ a \in dom(M_2) \end{cases}$$

Program Description. In FTAG, a program is described as a collection of module definitions and decompositions. Each such decomposition has the name of the module and the patterns into which the module is decomposed.

Definition 4.1 *The class $ActDef$ of activity definitions is defined by*

$$\begin{array}{lcl} Program & = & \mathcal{P}(Mdef) \\ Mdef & = & \begin{cases} M \Rightarrow D \\ \textbf{case}\ C_1 \Rightarrow D_1\ \ldots\ \textbf{otherwise} \Rightarrow D_{def}\ \textbf{end} \end{cases} \end{array}$$

M is a module, including its inherited and synthesized attributes, C_i is a condition, and D_i is a decomposition. The attributes of M are denoted by $Inh(M)$ and $Syn(M)$, respectively.

Definition 4.2 *The class D of decomposition patterns is one of the following*

$$D = \begin{cases} return \ \textbf{where} \ E \\ M_1 \dots M_k \ \textbf{where} \ E \\ \textbf{redo} \ M \ \textbf{with} \ M' \end{cases}$$

E is a collection of expressions e that define the relationship among attributes of M and $M_1, \dots, M_k$. It consists of the name of the defined attribute, a simple function for calculating the attribute and one or more attributes that are used as arguments:

$$e = (atr, func, args)$$

$atr : Atr$ is an attribute name, $func$ is a predefined function and $args$ are its arguments. The class of $func$ and $args$ are omitted when obvious from context.

Tree Structure. A tree structure is represented by a 3-tuple consisting of a root node, a set of nodes, and a mapping function.

Definition 4.3 *The class $Tree$ of generic trees is defined by*

$$\begin{array}{rcl} Tree & = & (r, N, C) \\ r & : & Node \\ N & : & \mathcal{P}(Node) \\ C & : & Node \times \textbf{int} \rightarrow Node \end{array}$$

r is a name of root node, N is a set of nodes, and C is a partial function that takes a node n and a number i indicating the ordering number of children, and returns the ith child of n. If $i \neq j$, then, of course $C(n, i) \neq C(n, j)$. If n has no children, the value of $C(n, i)$ is not defined; this is described by $C(n, i) = \perp$. $Node$ is a generic class of nodes and is treated as a primitive.

Computation Tree. The class of computation trees is a subclass of $Tree$, which is formalized as the following 9-tuple.

Definition 4.4 *The class CT of computation tree is defined by*

$$\begin{array}{rcl} CompTree & = & (r, N, C, A, M, I, S, P, V) \\ r & : & Node \\ N & : & \mathcal{P}(Node) \\ C & : & Node \times int \rightarrow Node \\ A & : & \mathcal{P}(Atr) \\ M & : & Node \rightarrow Module \\ I, S & : & Node \rightarrow \mathcal{P}(Atr) \\ P & : & \mathcal{P}(p) \ where \ p = (atr, func, args) \\ V & : & \mathcal{P}(v) \ where \ v = (atr, value), atr : Atr \end{array}$$

A is a set of attribute names; M is a function that maps a node n to the module name indicated by n; I and S are functions that map a node n to the set of its inherited and synthesized attributes, respectively; P is a collection of relationships among attributes; and V is a set of tuples consisting of an attribute name and its value. If no value has yet been calculated for atr, there is no corresponding tuple in V. Atr and $Module$ are the class of attribute and module names, respectively, and are treated as primitives. $atr : Atr$ is an instance of an attribute for which $atr \in A$ must be satisfied. $func$ and $args$ are also primitives, and are identical to those defined above.

Growth of a Computation Tree. A computation tree changes whenever a module decomposition is performed, with new nodes being added based on the particular decomposition. To do this, new nodes corresponding to the submodules are first created from the module decomposition. Each node has inherited and synthesized attributes, which are instantiated from the program script in order to make the name of the nodes and attributes unique in a given computation tree.

Definition 4.5 *For* $CT = (r, N, C, A, M, I, S, P, V) : CompTree, n_0 \in N, \forall i C(n_0, i) = \bot$,

$$
\begin{array}{l}
Grow(CT, n_0, \{M_1, \ldots, M_k\}) = \\
\quad (r, N \cup NewNode(M_1, \ldots, M_k), C', A \cup NewAttr(M_1, \ldots, M_k), M', I', S', P, V) \\
\quad where \quad
\begin{array}{lcl}
C'(n,i) & = & \begin{cases} n_i & if\ n = n_0 \\ C(n,i) & otherwise \end{cases} \\
M'(n) & = & \begin{cases} M_i & if\ n \in NewNode(M_1, \ldots, M_k) \\ M(n) & otherwise \end{cases} \\
I'(n) & = & \begin{cases} Inh_i & if\ n \in NewNode(M_1, \ldots, M_k) \\ I(n) & otherwise \end{cases} \\
S'(n) & = & \begin{cases} Syn_i & if\ n \in NewNode(M_1, \ldots, M_k) \\ S(n) & otherwise \end{cases}
\end{array}
\end{array}
$$

$NewNode(M_1, \ldots, M_k) = \{n_1, \ldots, n_k\}$ is a set of new nodes instantiated from $M_1, \ldots, M_k$. For each n_i, new attribute instances Inh_i and Syn_i are created, and $NewAttr(M_1, \ldots, M_k) = \bigcup_i Inh_i \cup Syn_i$, $M(n_i) = M_i$, $I(n_i) = Inh_i$,$S(n_i) = Syn_i$ are added to A, M, I, S, respectively.

Attribute Calculation. After new nodes and attributes are added to the computation tree, the new attribute values are calculated through the evaluation of the attribute equation. This operation is formally described by a function $Calculate : CompTree \times Node \times E \to CompTree$.

Definition 4.6 *For* $CT = (r, N, C, A, M, I, S, P, V) : CompTree, n_0 \in N$, *a function* $Calculate$ *is defined by*

$$
\begin{array}{l}
Calculate(CT, n_0, E) = (r, N, C, A, M, I, S, P \cup NewProduction(E), V \cup NewAtrValue(E)) \\
\quad where \\
\qquad \begin{array}{rcl}
NewProduction(E) & = & \bigcup_{e \in E}\ \{p \mid p = (atr, func, args)\} \\
NewAtrValue(E) & = & \bigcup_{e \in E}\ \{(atr, val) \mid evaluable(atr)\}
\end{array}
\end{array}
$$

For each attribute equation $e = (atr, func, args) \in E$, the equation $p = instance(e)$, made by instantiating atr and $args$ in e with corresponding attributes from A, is added to P. When p is ready to be evaluated, the value of atr is computed and the new tuple $v = (atr, val)$ is added to V.

Pruning a Computation Tree for Redoing. As long as only normal decompositions are performed, the computation tree grows monotonically. Redoing, however, reduces the tree by discarding some existing nodes. When a redoing decomposition has a **with** clause, which means a module M' different from M is to be executed, M is replaced by M'. Hence, redoing is realized by the following operations: (1) Pruning the subtree that contains the computation to be discarded, and (2) replacing the node if a **with** clause is specified.

Definition 4.7 *For* $CT = (r, N, C, A, M, I, S, P, V) : CompTree, n_0 \in N$*, a function* $Prune : CompTree \times Node \to CompTree$ *is defined by*

$$
\begin{array}{l}
Prune(CT, n_0) = CT' = (n, N - descendant(n_0), C', A - Attrof(descendant(n_0)), M', I', S', \\
\qquad P - \{p = (atr, func, args) \mid atr \in Attrof(descendant(n_0))\}, \\
\qquad V - \{v = (atr, val) \mid atr \in Attrof(descendant(n_0))\})
\end{array}
$$

$$
where \quad
\begin{array}{lcl}
C'(n,i) & = & \begin{cases} \perp & if\ n \in descendant(n_0) \\ \perp & if\ n = n_0 \\ C(n,i) & otherwise \end{cases} \\
M'(n) & = & \begin{cases} \perp & if\ n \in descendant(n_0) \\ M(n) & otherwise \end{cases} \\
I'(n) & = & \begin{cases} \perp & if\ n \in descendant(n_0) \\ I(n,i) & otherwise \end{cases} \\
S'(n) & = & \begin{cases} \perp & if\ n \in descendant(n_0) \\ S(n,i) & otherwise \end{cases} \\
Attrof(ns) & = & \bigcup_{n \in ns}(S(n) \cup I(n))
\end{array}
$$

$descendant(n)$ denotes a set of n's descendant nodes; more precisely, $descendant(n) = \{m \mid \exists i; C(n,i) = m \ \vee \ \exists i, m' \in descendant(n); C(m', i) = m\}$. Note that $n_0 \notin descendant(n_0)$. CT' is a tree in which all nodes under n_0(except n_0) have been removed.

Definition 4.8 *For* $CT = (r, N, C, A, Mod, I, S, P, V) : CompTree, n_0 \in Node, M' \in Module$*, a function* $Replace : CompTree \times Node \times Node \times Module$*, which replaces the node and module with another node and module, is defined by*

$$
Replace(CT, n_0, n_0', M') = (r, N, C', A, Mod', I, S, P, V)
$$

$$
where \quad
\begin{array}{lcl}
C'(n,i) & = & \begin{cases} n_0' & if\ C(n,i) = n_0 \\ C(n,i) & otherwise \end{cases} \\
Mod'(n) & = & \begin{cases} M' & if\ n = n_0 \\ Mod(n) & otherwise \end{cases}
\end{array}
$$

Recall that the number and type of n_0 and n_0''s attributes must be identical, so there are no effects to any attribute instances or values except the node name.

Execution Semantics. Execution is the process of transforming computation trees by decomposition and attribute calculation. Each step of the transformation can be described formally as a transform function $Exec : CompTree \times Node \times D \to CompTree$. D denotes the class of decompositions shown in Definition 4.2. There are three cases to be considered.

Case (1): Normal Decomposition. If D is a normal decomposition, $Exec$ is defined as follows:

Definition 4.9 *For* $CT = (r, N, C, A, M, I, S, P, V) : CompTree, n_0 \in N, \forall i C(n_0, i) = \perp$,

$$
\begin{array}{l}
Exec(CT, n_0, \{return\ \mathbf{where}\ E\}) = Calculate(CT, n_0, E) \\
Exec(CT, n_0, \{M_1, \ldots, M_k\ \mathbf{where}\ E\}) = Calculate(Grow(CT, n_0, \{M_1, \ldots, M_k\}), n_0, E)
\end{array}
$$

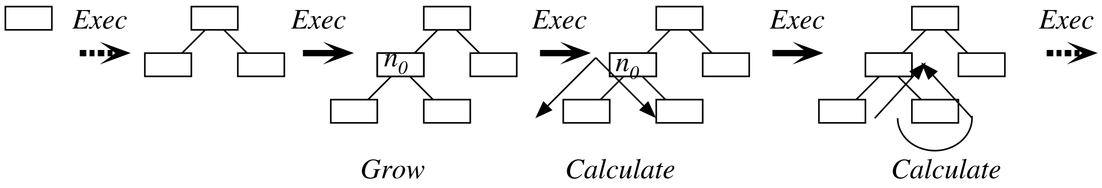

Figure 9: Execution by Series of Tree Transformations

If n_0 is a primitive, synthesized attribute values are calculated by the function $Calculate$; otherwise n_0 should be decomposed into submodules. The decomposition is performed by the function $Grow$, with attributes being calculated by $Calculate$.

Case (2): Redoing Decomposition. When D is a redoing decomposition, it is performed by pruning the computation tree. When a **with** clause is specified, a node replacement will also occur.

Definition 4.10 *For* $CT = (r, N, C, A, Mod, I, S, P, V) : CompTree, n_0 \in N, \forall i C(n_0, i) = \perp$,

$$Exec(CT, n_0, \{\ \mathbf{redo}\ M\ \}) = Prune(CT, m)$$
$$Exec(CT, n_0, \{\ \mathbf{redo}\ M\ \mathbf{with}\ M'\ \}) = Replace(Prune(CT, m), m, m', M')$$

m is the most recent occurrence of $Mod(m) = M$, determined statically by an analysis of the program. m' is a new node corresponding to M'.

Case (3): Execution Sequence. The program is executed by repeated applications of $Exec$, starting with an initial tree containing only a root node. Figure 9 illustrates this process; $Grow$ adds new nodes to the tree, while $Calculate$ determines the values associated with each node.

5 Implementing FTAG

In this section, we outline how the value reference model described in the previous sections can be implemented in a standard computer system consisting of multiple processors and (logically) shared external storage (e.g., disk). Specifically, we address the issues of processors and module allocation, and the use of external storage to implement a stable object base for storing attribute values. Optimization issues are also touched upon briefly.

5.1 Processors and Module Allocation

Each processor is assumed to consist of a computing engine and a mechanism for sending and receiving messages. We abstract this message-passing mechanism as a $port$, where a port is a pair of synchronous communication channels. Thus, a value written to a send port is read by another processor from the corresponding receive port. Each processor has two sets of attributes that represent the send and receive ports, respectively. Formally,

$$\begin{array}{l} Proc = \mathcal{P}(proc) \\ proc = (Send, Recv) \\ \quad where\ Send, Recv : \mathcal{P}(Atr) \end{array}$$

$Proc$ is a set of processors $proc$, while $Send$ and $Recv$ are each a set of attributes that are mapped to the processor's send and receive ports. A value assigned to a $Send$ port attribute is sent automatically to the $Recv$ port on another processor to which the same attribute has been mapped.

Processors are assigned to each node in the computation tree, and are responsible for all the communication between the node and its children. Information is passed between processors using the port mechanism. This process of assigning nodes to processors is described formally by partitioning N, A, P, V in $CompTree$. Such a partition is defined by a total function that maps a processor to the set of nodes assigned to that processor, $Ass : proc \to \mathcal{P}(Node)$. For simplicity, N_i is taken to be synonymous with $Ass(p_i)$, that is, the set of nodes assigned to the processor p_i. Of course, $\bigcup_i N_i = N$. $A_i = \{Inh(n) \cup Syn(n) \mid \forall n; n \in N_i\}$ means the set of attribute instances assigned to p_i. P_i and V_i are defined in the same manner.

A node is assigned to a processor when it is created. This is performed by changing the assignment map Ass whenever $Grow$ is applied. Attribute values may be passed between a node and its children during an attribute calculation. Values destined for a node assigned to another processor must be passed to appropriate processor. Such attributes must be connected to ports, and must be included in $Send$ and $Recv$ of the appropriate processors. This is performed by changing the processor's $Send$ and $Recv$ set during the application of $Calculate$. Formally, assume $p_i, p_j \in Proc, CT = (r, N, C, A, M, I, S, P, V), A_i = Ass(p_i), A_j = Ass(p_j)$. Then, whenever $Calculate(CT, n_0, E)$ is applied, $P_{i,j}$,$Send_{i,j}$ and $Recv_{i,j}$ must be changed for each evaluation of $e = (atr, func, args) \in E$ according to the following rules:

(1) If $atr \in A_i \wedge$ all args are in $A_i \Rightarrow$ add e to P_i
(2) If $atr \in A_j \wedge$ all args are in $A_j \Rightarrow$ add e to P_j
(3) If $atr \in A_j \wedge$ all args are in $A_i \Rightarrow$ add e to P_i and atr to $Send(p_i)$ and $Recv(p_j)$
(4) If $atr \in A_i \wedge$ all args are in $A_j \Rightarrow$ add e to P_j and atr to $Send(p_j)$ and $Recv(p_i)$

Assignment of $\perp$ to attributes of modules that were being executed on processors that suffered operational failures is done using the port mechanism. A timer is set when a reply from a module is expected, and its attribute values set to $\perp$ should the expire with no reply being received. This technique is, of course, just the standard use of timeouts for failure detection.

5.2 Stable Object Base

In the value reference model, no specific mention is made of where attribute values are actually stored during a computation. One simple and efficient option is in primary memory, but this is appropriate only for those programs for which the amount of space needed is relatively small. A more realistic choice is to store the values on secondary storage using an abstraction that we will refer to as a $stable\ object\ base$. As a form of stable storage, values stored in this object base are assumed to survive failures.

Given the greater access times for the object base relative to primary memory, it is important to store only those values that are required to guarantee continued execution. For FTAG programs, there are two such sets of values: (1) all inherited and synthesized attributes that are needed for the current calculation, and (2) all inherited attributes that may be needed for redoing purposes. We call (1) the set of $vital\ values$. Vital values change as the computation progresses; for example, if a module is decomposed and computation is moved to submodules, the inherited attributes of the module are no longer vital. The set (2) includes all inherited attributes of modules in recovery blocks or those that may be the target of rollbacks following operational failures.

In order to formally characterize the set of vital values, we use $\Omega : Atr \to Obj$ to describe the mapping from attributes to the object base. The notation $domain(\Omega)$ is used to refer to the set of attributes stored in the object base. Hence, the addition of an attribute to $domain(\Omega)$ implies the allocation of new storage space for that attribute, while its deletion means destruction of the value and release of associated storage.

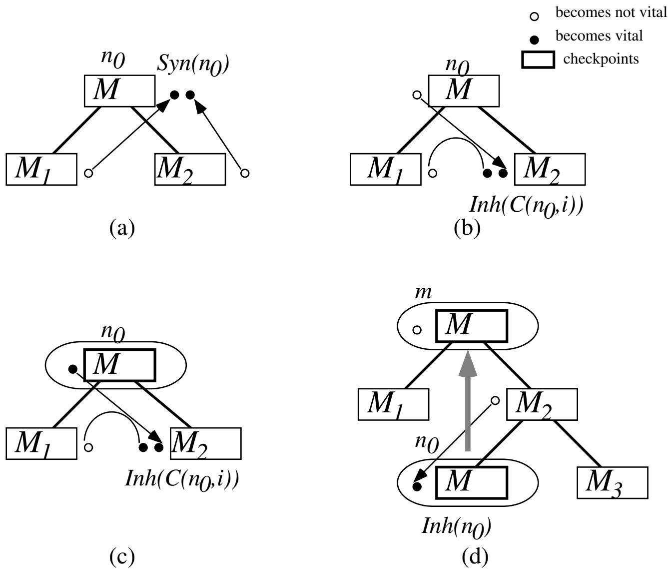

Figure 10: Rules for Storage Mapping

Changes to $domain(\Omega)$ can be defined step by step for each application of $Grow$ or $Calculate$. The actions taken, however, depend on whether a given module might be redone, so we first define $\mathcal{M}$ as the set of such modules:

Definition 5.1 *$\mathcal{M}$ contains all modules M that appear in $\{$**redo** $M\}$ decompositions.*

Call such an $M \in \mathcal{M}$ a *checkpoint module*. Note that such modules can be determined statically from the text of the program.

When an attribute calculation is performed by $calculate(CT, n_0, E)$, the following rules are applied for each $e = (atr, func, args) \in E$,

(a) After all $atr \in Syn(n_0)$ have been calculated, all $Inh(n_0)$ and $Syn(C(n_0, i))$ used as arguments are no longer vital.
(b) If $atr \in Inh(C(n_0, i))$ and $n_0 \notin \mathcal{M}$, all $Inh(n_0)$ and $Syn(C(n_0, j))$ used as arguments are no longer vital.
(c) If $atr \in Inh(C(n_0, i))$ and $n_0 \in \mathcal{M}$, all $Inh(n_0)$ must be kept in the object base.
(d) If $atr \in Inh(n_0)$, $n_0 \in \mathcal{M}$ and n_0 overrides m, $Inh(n_0)$ replaces $Inh(m)$.

In (d), the node n_0 is said to *override* m if n_0 and m are instantiations of the same checkpoint module, and m is an ancestor of n_0.

Figure 10 illustrates these rules. (a) denotes the point at which computation is going upwards; in this case, none of the attributes of n_0's children are needed unless one or both are checkpoint modules. (b),(c) and (d) shows different situations when the computation is flowing down. If there are no checkpoint modules among n_0 and $C(n_0, i)$, no special treatment is needed; otherwise, the appropriate values must be kept. Also, note that in (d), m is overridden by n_0 because both are instantiations of M and m is an ancestor of n_0.

Finally, we point out that it might be worthwhile to retain attribute values that would normally be overwritten in the object base. For example, this would allow failures to be handled even if they require that the computation be rolled back further than the most recent instantiation of a checkpoint module. To do this would require $object\ versioning$ with the following characteristics:

(1) Whenever a new value is stored, it is checked in as a new version.
(2) Whenever redoing occurs, the most recent version is retrieved.
(3) Older versions are retrieved using an explicit mechanism.

With this scheme, only the most recent version is "visible" within the standard computation model, as specified by (1) and (2). (3) allows divergence from this model when necessary to retrieve older versions to cope with failures that require more extensive rollback. Work is just beginning on designing and implementing a versioning object base along the lines of this design.

5.3 Optimizations

Redoing is inherently an expensive operation, but with a functional model such as FTAG, optimizations are often possible based on module execution order. For example, consider the following:

$$M \Rightarrow M_1(x \mid u) \;\; M_2(x \mid v) \;\; M_3(u, v \mid y)$$
$$M_3(u, v \mid y) \Rightarrow \ldots \mathbf{when}\ fail(v) \Rightarrow \mathbf{redo}\ M(x \mid u) \ldots$$

Assume that a failure is detected during the execution of M_3 that requires redoing M. Since the bad value is v, the failure must have occurred during the execution of module M_2, which generates v. Hence, although the target of redoing is M, only M_2 and M_3 need be redone. Such data dependencies can be determined statically, so that it would be possible to keep an internal table listing exactly which modules need be redone for each module that is used in a redoing decomposition.

Specification of an explicit computation order can also be exploited for optimization purposes in certain cases. For example, assume that M_2 is a checkpoint module in $\{M_1; M_2; M_3\}$. This ordering means that M_3 will be reexecuted if M_2 is ever redone, thereby allowing M_3's inherited attributes to be discarded to save storage space. In contrast, if no ordering were specified here, the inherited attributes of all three modules would have to be stored since they could potentially be needed for redoing. Of course, a dataflow analysis of the program could expose this optimization in the implicit case as well, but only at significant cost.

6 Conclusions

FTAG is a computational model that is well-suited for writing fault-tolerant software due to its functional nature and the inherent ease with which actions can be redone should failures occur. Although others have noted similar advantages with respect to functional programming [HNS89, JA91], we believe that ours represents one of the first attempts to develop a formal computational model based on these ideas. Moreover, as noted in the Introduction, this model is derived from similar attribute based models that have proved useful is other areas of computer science, such as object-oriented programming and the description of high-level software design processes.

Future work will concentrate in two different areas. One is implementing a programming system that will allow execution of programs based on the FTAG model. Such a realization will be based on either a distributed or multiprocessor architecture. The other is extending the model to incorporate other common fault-tolerance paradigms such as replication. The goal here would be to determine whether characteristics of our model offer advantages for active redundancy similar to those illustrated here for rollback-oriented techniques. In both cases, our efforts will include investigating realistic applications to test the true benefits of this approach.

References

[Avi85] A. Avizienis. The N-Version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[HNS89] R. Harper, , G. Nagle, and M. Serrano. Use of a functional programming model for fault tolerant parallel programming. In *Proceedings of the Nineteenth Symposium on Fault-Tolerant Computing*, pages 20–26, Chicago, IL, Jun 1989.

[JA91] R. Jagannathan and E. Ashcroft. Fault tolerance in parallel implementations of functional languages. In *Proceedings of the 21st Symposium on Fault Tolerant Computing*, pages 256–263, Montreal, Canada, Jun 1991.

[Kat81] T. Katayama. HFP, a hierarchical and functional programming based on attribute grammars. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 343–353, 1981.

[Kat89] T. Katayama. A hierarchical and functional software process description and its enaction. In *Proceedings of the Eleventh International Conference on Software Engineering*, pages 343–352, 1989.

[Knu68] D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Lam81] B. Lampson. Atomic transactions. In *Distributed Systems—Architecture and Implementation*, pages 246–265. Springer-Verlag, Berlin, 1981.

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

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, Jun 1975.

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

[SK88] Y. Shinoda and T. Katayama. Attribute grammar based programming and its environment. In *Proceedings of the 21st Hawaii International Conference on System Sciences*, pages 612–620, Kailu-Kona, Jan 1988.

[SK90a] Y. Shinoda and T. Katayama. OOAG: An object-oriented extension of attribute grammar and its implementation using distributed attribute evaluation algorithm. In *Proceedings of WAGA, International Workshop on Attribute Grammar and its Application*, volume 461, pages 177–191. LNCS Springer-Verlag, 1990.

[SK90b] M. Suzuki and T. Katayama. Redoing: A mechanism for dynamics and flexibility of software processes. In *Proceedings of InfoJapan '90*, pages 151–160, Tokyo, Japan, 1990.