FTAG: A Functional and Attribute Based Model for Writing Fault-Tolerant Software

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

TR 96-6

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 (Fault-Tolerant Attribute Grammars). The basic model is introduced first, followed by a description of mechanisms that allow a variety of standard fault-tolerance techniques to be realized in a straightforward way. Techniques that can be accommodated include replication and checkpointing to tolerate operational faults, and recovery blocks and N-version programming to tolerate software faults. Several examples are given to illustrate these techniques, including a replicated name server and a fault-tolerant sort that uses recovery blocks. A formal description of FTAG that precisely specifies the semantics of the model is also presented. Finally, a software architecture describing how FTAG can be implemented in a computer system containing multiple processors is given.

May 21, 1996

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., 1-1 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* [1] and *N-version programming* [2] for dealing with software faults, and *checkpointing* [3], *atomic actions* [4], and the *replicated state machine approach* [5] 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 (Fault-Tolerant Attribute Grammars), 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 [6] for such diverse purposes as functional programming [7, 8], object-oriented programming [9], and modeling of the software development process [10]. This approach offers several advantages, including a declarative style, separation of semantic and syntactic definitions, and the simplicity of a functional foundation. Others have also recognized the attraction of a functional model for this type of programming [11, 12, 13, 14], although FTAG differs in how it provides support for fault tolerance and in its use of an attribute grammar formalism.

This paper is organized as follows. An introduction of FTAG is given in section 2, with the basic model being described first, followed by features like *redoing* [15] and *replication* that provide the fundamental basis for implementing recovery, N-version programming, and similar mechanisms. This is followed in section 3 by a description of how this model can be used for fault-tolerance; several examples are given to illustrate the approach, including one based on recovery and another that uses replication. In section 4, a formal definition and operational semantics of FTAG are provided. Section 5 then gives a software architecture for implementing FTAG on loosely-coupled multi-processor systems, including both storage management and processor allocation mechanisms. Finally, section 6 discusses the advantages of the approach and related work, while section 7 contains conclusions and directions for future work.

2 An Overview of the FTAG Computational Model

In this section, we summarize the FTAG computational model introduced in [16]. The model is based on the HFP (Hierarchical and Functional Process) model [7], which is in turn derived from attribute grammars. The basics of the model are outlined first, followed by a description of facilities for fault-tolerant software. The full grammar for FTAG is given as an Appendix.

2.1 The Basic 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:

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

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.

$$\begin{array}{lll} \multicolumn{3}{l}{M(x_1, \ldots, x_n | y_1, \ldots, y_m) \Rightarrow} \\ [& C_1 & \rightarrow D_1 \\ | & \vdots & \\ | & C_n & \rightarrow D_n \\ | & \textbf{otherwise} & \rightarrow D_{def} \quad] \end{array}$$

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. Note that, in a system consisting of multiple processors, each module can be allocated to a separate processor to be executed, assuming that its inherited attributes are available. An allocation scheme that exploits this characteristic is described further in section 5.

Figure 1 shows a part of 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, a factor that further enhances its suitability for 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:

$$\begin{array}{ll} \multicolumn{2}{l}{\textbf{otherwise} \rightarrow} \\ \quad \{less\_half(\hat{}seq.in, \tilde{}seq.in|seq.less); & \\ \quad \quad grtr\_half(\hat{}seq.in, \tilde{}seq.in|seq.grtr)\} & (1;2) \\ \quad qsort(seq.less|seq.lessout) & (3) \\ \quad qsort(seq.grtr|seq.grtrout) & (4) \end{array}$$

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

qsort(seq.in | seq.out) ⇒
  [   (seq.in ==<>) or
      (~seq.in ==<>) → return
      where
       seq.out = seq.in
   |  otherwise →
       less_half(^seq.in, ~seq.in | seq.less)     (1)
       grtr_half(^seq.in, ~seq.in | seq.grtr)     (2)
       qsort(seq.less | seq.lessout)              (3)
       qsort(seq.grtr | seq.grtrout)              (4)
      where
       seq.out = seq.lessout
        + < ^seq.in > +seq.grtrout  ]
less_half(x.ref, seq.in|seq.out) ⇒
  [   seq.in ==<>→ return
      where
       seq.out = seq.in
   |  ^seq.in >= x.ref →
       less_half(x.ref, ~seq.in|seq.out)
      where
       (* No attribute equations *)
   |  ^seq.in < x.ref →
       less_half(x.ref, ~seq.in|seq.subout)
      where
       seq.out =< ^seq.in > +seq.subout
   |  otherwise → return
      where
       seq.out = seq.in  ]
grtr_half(x.ref, seq.in|seq.out) ⇒
  [   seq.in ==<>→ return
      where
       seq.out = seq.in
   |  ^seq.in < x.ref →
       grtr_half(x.ref, ~seq.in|seq.out)
      where
       (* No attribute equations *)
   |  ^seq.in >= x.ref →
       grtr_half(x.ref, ~seq.in|seq.subout)
      where
       seq.out =< ^seq.in > +seq.subout
   |  otherwise → return
      where
       seq.out = seq.in  ]
```

Figure 1: Quicksort in FTAG

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.

Another feature for describing explicit ordering is decomposition to the collection of same module with different inputs, callled $iterative\ decomposition$, generally described as follows:

$$\begin{aligned}&\mathbf{const}\ N\ = 100\ \ (*number\ of\ iterations*)\\ &\mathbf{type}\ V\ \ = \mathbf{array}\ 1..N\ \mathbf{of}\ v\end{aligned}$$

$$M_{body}(i \mid v) \Rightarrow ...(*body\ of\ calculation*)$$

$$M(\mid V) \Rightarrow \mathbf{for}\ i = 1..N\ \ \mathbf{do}\ \ M_{body}(i \mid v)\quad \mathbf{where}\ \ V[i] = v$$

N is a constant which represents the number of iterations. For each enactment of M_{body}, i is passed as an input and results are stored and collected in array V whose element type is v. It is useful for describing a calculation such that the number of iteration is already given.

2.2 Fault-Tolerance Features

Redoing. $Redoing$ is an operation that replaces a portion of the computation tree with a new computation. Although redoing has many uses, in the context of fault-tolerant software we use it 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. We assume that all failures of interest are manifested by incorrect attribute values

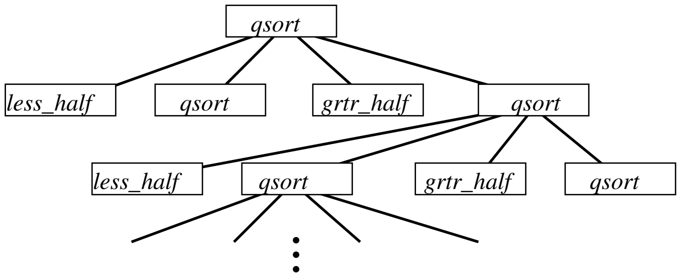

Figure 2: Quicksort Computation Tree

that can be tested by a conditional. Such an assumption is common for software faults [1], while missing or incorrect attribute values caused by operational faults such as crashed processors can be translated into a distinguished value $\perp$ ("bottom") that is assigned to the appropriate attributes by the underlying system.

As an example of redoing, assume that 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 re-executed. 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 above example can be described in FTAG as follows:

$$\begin{array}{lcl} M(x \mid y) & \Rightarrow & M_1(x_1 \mid y_1)\ \ M_2(x_2 \mid y_2)\ \ M_3(x_3 \mid y_3)\ \ \textbf{where}\ \ldots \\ M_3(x \mid y) & \Rightarrow & [\ \ valid(x)\ \ \rightarrow\ \ M_3body(x \mid y) \mid\ \ \textbf{otherwise}\ \ \rightarrow\ \ \textbf{redo}\ M\ \ ] \end{array}$$

In M_3, the condition $valid$ tests the value of input attribute x to verify that a failure has not occurred. 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 3 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.

Replication. Replicating software and executing it in parallel is another standard approach to writing software that can tolerate software and/or operational faults [2, 5]. FTAG can be used to realize such replication in a straightforward and intuitive way. In fact, no syntactic changes are needed, only an extension of the interpretation to allow the submodules into which a module is to be decomposed to have the same set of inherited and synthesized attributes. For example,

$$M(x \mid y) \Rightarrow M_1(x \mid y)\ M_1(x \mid y)\ M_1(x \mid y)$$

decomposes M into three identical modules M_1, each of which has the same inherited and synthesized attributes, x and y, respectively. This type of decomposition is termed a *replicated decomposition*, and

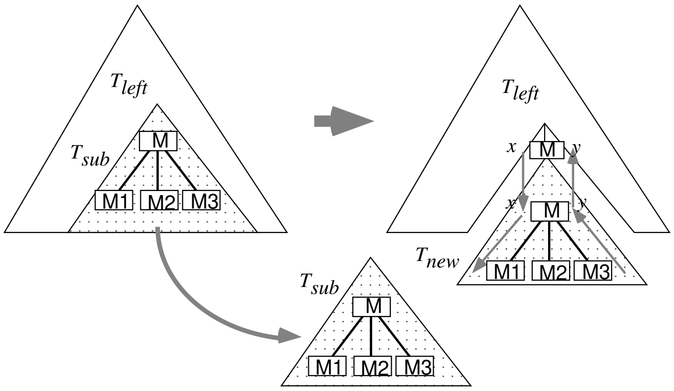

Figure 3: A Computation Tree with Redoing

the M_1 are called $replicas$. The interpretation of replicated decomposition is that each replica is invoked simultaneously as in a normal decomposition, but with only one of the generated results used as the output of M. Module names need not be identical, which facilitates programs that use N-version programming.

Typically, only one result is selected for use from the collection of replicated modules. This could be done depending on the specifics of the application by either selecting any non-bottom value or by using a general collation function.

Stable Object Access. Attribute values in this model are either stored in primary memory or in a *stable object base* depending on whether or not they might be needed after a failure [17]. Certain activities such as redoing are also facilitated if attribute values in the object base that would normally be overwritten are retained instead. To do this, the model is extended to support $object\ versioning$ and the definition of attribute value extended to support a complex structure that contains some old values. We call such an object a *versioned object*, and the extended model the *versioned object model*.

Values that represent versioned objects are denoted by explicitly placing a '*' before their name, with only the most recent version being "visible" within the standard computation model. That is, the most recent value is used as the attribute value whenever that attribute is referenced. Thus, a versioned object can be used as follows:

$$M(m, n, *obj.in \mid\ *obj.out) \Rightarrow$$
$$M_1(e, obj.in|obj.out)\ \mathbf{where}\ e = f(m, n)$$

The value of "obj.in" is retrieved from the object base before the computation of the module, while the new value "obj.out" is stored as the newest version after the computation completes. Note that versioned objects always appear with modifiers since both before and after values are used whenever such objects are referenced within a computation.

Versioned objects can also be used in checkpointing and replication. This is discussed further in section 5.1.

3 Fault-Tolerant Programming using FTAG

In this section, we demonstrate using several examples how various fault-tolerance techniques can be expressed naturally using FTAG.

3.1 Recovery using Redoing

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 [1], and *checkpointing*, a technique used to recover from operational faults [3].

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 can be realized using redoing in FTAG as follows:

$$
\begin{array}{l}
M(x \mid y) \Rightarrow try\_M_1(x \mid y) \\
\\
try\_M_1(x \mid y) \Rightarrow \{M_1(x \mid y); \\
\qquad\qquad [AT_1(y) \rightarrow return \\
\qquad\qquad \mid \mathbf{otherwise} \rightarrow \mathbf{redo}\ try\_M_1\ \mathbf{with}\ try\_M_2\ \ ]\ \} \\
\\
try\_M_2(x \mid y) \Rightarrow \{M_2(x \mid y); \\
\qquad\qquad [AT_2(y) \rightarrow return \\
\qquad\qquad \mid \mathbf{otherwise} \rightarrow \mathbf{redo}\ try\_M_2\ \mathbf{with}\ try\_M_3\ \ ]\ \} \\
\qquad\qquad\qquad \vdots \\
try\_M_k(x \mid y) \Rightarrow M_k(x \mid y)
\end{array}
$$

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 4 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

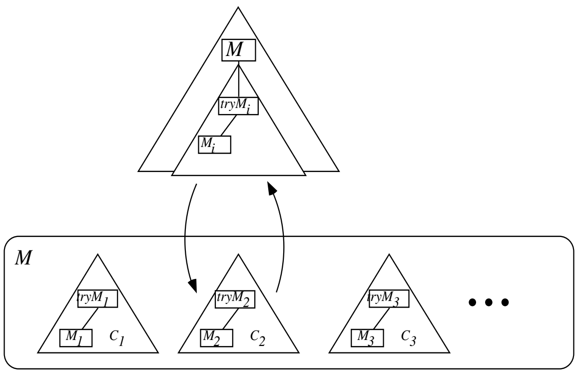

Figure 4: Computation Tree for Recovery Blocks

improve readability, the following syntax can also be used:

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

As a more concrete example, consider using recovery blocks to sort numbers into ascending order, as shown Figure 5. 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:

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

Figure 5: Using Recovery Blocks for Sorting

$$
\begin{array}{l}
grtr\_half(x.ref, seq.in|seq.out) \Rightarrow \\
\quad [\quad\quad\quad\quad \vdots \\
\quad \mid\ \hat{}\,seq.in > x.ref \rightarrow (*\ Error\ !!\ *) \\
\quad\quad\quad grtr\_half(x.ref, \tilde{}\,seq.in|seq.subout) \\
\quad\quad \textbf{where}\ seq.out = <\ \hat{}\,seq.in > + seq.subout \\
\quad\quad\quad\quad\quad \vdots
\end{array}
$$

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* [18] as follows by having M_{22} check its input attribute and redo M_2 should it have the value $\perp$.

$$
\begin{array}{l}
M_{22}(v \mid u) \Rightarrow \\
\quad [\ v \neq \perp \qquad \rightarrow \textbf{redo}\ M_2 \\
\quad \mid \textbf{otherwise} \ \ \rightarrow M_{22}body(v \mid u)\ ]
\end{array}
$$

To illustrate this use of FTAG further, consider the outline of a long running scientific application shown in Figure 6. This program calculates a large table of numbers, where the cost of calculating each $V[i]$

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

Figure 6: An Expensive Calculation without Redoing

$$
\begin{array}{l}
main(\ \mid V) \Rightarrow \textbf{for}\ i = 1..N \quad Calc(i \mid v)\ \textbf{where}\ V[i] = v \\
\\
Calc(i \mid v) \Rightarrow \{calc(i \mid v); \\
\qquad\qquad\qquad [\ v = \perp \rightarrow\ \textbf{redo}\ Calc \\
\qquad\qquad\qquad \mid \textbf{otherwise} \rightarrow\ return\ \ ]\,\}
\end{array}
$$

Figure 7: Computation modified to use Redoing

is 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 7. 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 the stable object base outlined above. We refer to these values in the object base as a *checkpoint*, although it is important to emphasize that only the inherited attributes need be stored and not the entire state of the module as is often the case when the term is used. Note also that it would be possible to structure this calculation so that checkpoints are established every i iterations to minimize use of this stable object base rather than every iteration as done here.

3.2 Building a Name Server using Replication

Next, we illustrate the way in which replication can be used in FTAG by describing a name server that uses replicated decompositions to realize its functionality. In addition to serving as a substantial example of

FTAG's replicated decomposition construct, this program illustrates how tolerance of both operational and software faults can easily be combined into a single program that uses both recovery and replication.

The overall architecture of this name server is shown in Figure 8. *NS* is the name server's main module,

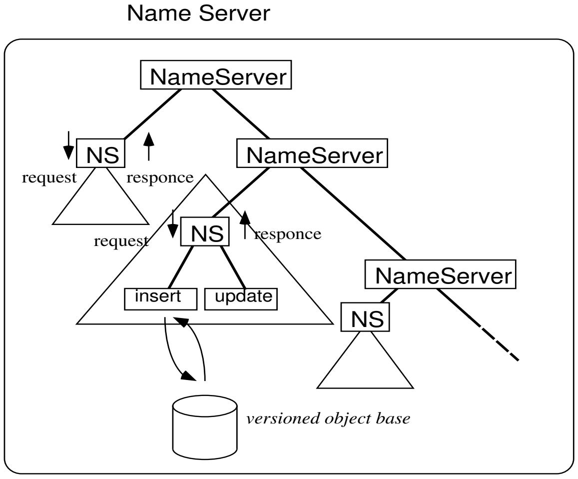

Figure 8: Name Server architecture

which in turn invokes submodules that implement the following operations:

- lookup($name$): Return the address of $name$.
- update($name, addr$): Update entry $name$ to reflect new address.
- delete($name$): Delete entry $name$.

To realize the name serve, the versioned objects introduced in section 2.2 have to be used to store the bindings between names and addresses. This is accomplished as follows:

$$
\begin{array}{l}
Update(name, addr \mid res) \Rightarrow \\
\quad do\_update(name, addr, *bind.in \mid res, *bind.out)
\end{array}
$$

The value of "bind.in" is retrieved from the object base before the computation of the module *do_update*, while the new value "bind.out" is stored as the newest version after the computation completes. Figure 9 illustrates this concept.

The program script is shown in Figure 10 and 11. For simplicity, each request is performed sequentially. $NameServer$ in turn invokes NS; this non-replicated module returns $\perp$ if an operational failure occurs during its execution, in which case redoing is used as a recovery mechanism. An acceptance test ($LookUp$) is also performed after execution of the $Update$ or $Delete$ modules to detect software faults; if the test fails, redoing is used. Here, NS is redone for conciseness, although in practice, another version of the module would likely be invoked to avoid the original problem. $LookUp$, which is the module responsible for lookup operations, using replicated decomposition to execute three replicas of do_lookup. Before invoking each replica, $LookUp$ retrieves the $bind.in$ from the object base and passes it as the second inherited attribute. If one or two of the replicas fail, the correct result is given; if, however, none of the replicas provides a result, the value of output res is considered to be $\perp$ and redoing occurs in *NS* as described above. Note that the attribute $bind.tmp$ is not preceded by '*', meaning that this value is discarded after the computation. The program scripts for $Update$ and $Delete$ are analogous.

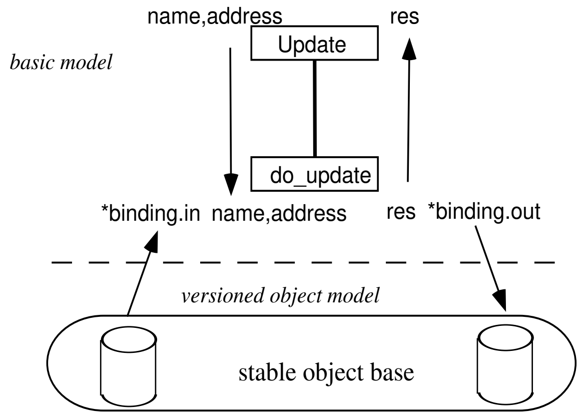

Figure 9: Objectbase Access

4 Formal Description

In this section, we make FTAG more precise by providing a formal description of the model and its operational semantics. We focus first on the basic model, and then discuss fault-tolerance features including redoing and replication.

4.1 Preparation

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: The set of all subsets of a set A is called the $powerset$ 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. Domain Restrictions: When a mapping $M : A \rightarrow B$ is given, the function whose domain is restricted to $R \subseteq A$ is denoted by $R \lhd M$. More precisely, $R \lhd M = \{x \mapsto y \mid x \in R, y = M(x)\}$. Domain restriction can be applied on tuples. If $S = (X_1, X_2, \ldots)$, $R \lhd S$ is defined as $(R \lhd X_1, R \lhd X_2, \ldots)$.

5. 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 $a : A$, $M = M_1 \cup M_2$,

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

4.2 Semantics of the Basic Model

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.

$$
\begin{array}{l}
NameServer(req|res) \Rightarrow \\
\quad NS(req|res) \\
\quad [\, res == \bot \quad \rightarrow \ \textbf{redo}\ NS(req \mid res) \\
\quad \mid \textbf{otherwise} \ \rightarrow \ return \ ] \\
\\
NS(req \mid res) \Rightarrow \\
\quad [\, optype(req) == \text{"}lookup\text{"} \rightarrow LookUp(name(req) \mid res) \\
\quad \mid optype(req) == \text{"}update\text{"} \rightarrow \\
\qquad Update(name(req), address(req) \mid res) \\
\qquad LookUp(name(req) \mid res.tmp) \\
\qquad\ [\, restype(res.tmp) == \text{"}Notfound\text{"} \rightarrow \textbf{redo}\ NS(req \mid res) \\
\qquad\ \mid \textbf{otherwise} \rightarrow return \ ] \\
\quad \mid optype(req) == \text{"}delete\text{"} \rightarrow \\
\qquad Delete(name(req) \mid res) \\
\qquad LookUp(name(req) \mid res.tmp) \\
\qquad\ [\, restype(res.tmp) \neq \text{"}Notfound\text{"} \rightarrow \textbf{redo}\ NS(req \mid res) \\
\qquad\ \mid \textbf{otherwise} \rightarrow return \ ] \\
\quad \mid \textbf{otherwise} \rightarrow return \\
\quad\ \textbf{where}\ res = makeres(\text{"}WrongOperation\text{"}) \ ]
\end{array}
$$

Figure 10: Name Server (top level)

Definition 4.1 *The class $MDef$ of module definitions is defined by*

$$Mdef \ \stackrel{def}{=} \ \{ \ M \Rightarrow [\ C_1 \rightarrow D_1 \mid \ \ldots \mid \textbf{otherwise} \rightarrow D_{def} \ ]$$

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. If the decomposition is unconditional, we consider it as defined in D_{def}.

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

$$D \stackrel{def}{=} \begin{cases} return \ \textbf{where} \ E \\ M_1 \ldots M_k \ \textbf{where} \ E \\ \textbf{for} \ R \ \textbf{do} \ M' \ \textbf{where} \ E \\ \textbf{redo} \ M \end{cases}$$

E is a collection of expressions e that define the relationship among attributes of M and $M_1, \ldots, M_k$. 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.

$$
\begin{array}{l}
LookUp(name \mid res) \Rightarrow \\
\quad do\_lookup(name, *binding.in \mid res, *binding.out) \\
\quad do\_lookup(name, *binding.in \mid res, *binding.out) \\
\quad do\_lookup(name, *binding.in \mid res, *binding.out) \\
\\
do\_lookup(name, binding.in \mid res, binding.out) \Rightarrow \\
\quad [\ binding.in == <> \rightarrow return \\
\qquad \textbf{where}\ \ binding.out = binding.in \\
\qquad\qquad res = makeres(\text{"}NotFound\text{"}) \\
\quad \mid name(\hat{}binding.in) == name \rightarrow return \\
\qquad \textbf{where}\ \ binding.out = \tilde{}binding.in \\
\qquad\qquad res = makeres(address(\hat{}binding.in), \text{"}Found\text{"}) \\
\quad \mid \textbf{otherwise} \rightarrow \\
\qquad\quad do\_lookup(name, \tilde{}binding.in \mid res, binding.tmp) \\
\qquad \textbf{where}\ \ binding.out = \hat{}binding.in + binding.tmp\ \ ] \\
\\
Update(name, address \mid res) \Rightarrow \ldots\ analogous\ to\ lookup\ \ldots \\
\\
do\_update(name, address, binding.in \mid res, binding.out) \Rightarrow \\
\quad do\_lookup(name, binding.in \mid res.tmp, binding.tmp) \\
\quad\ [\ restype(res.tmp) == \text{"}Notfound\text{"} \rightarrow \\
\qquad\quad do\_insert(name, address, binding, in \mid res, binding.out) \\
\quad\ \ \mid \textbf{otherwise} \rightarrow return \\
\qquad\ \ \textbf{where}\ \ binding.out = <\ (name, address) > + binding.tmp \\
\qquad\qquad\ \ res = makeres(name, address, \text{"}Success\text{"})\ \ ] \\
\\
\quad \ldots
\end{array}
$$

Figure 11: Name Server (replicated operations)

Computation Tree. The status of the computation can be represented by a computation tree, which consists of nodes corresponding to an enactment of module decomposition. In general, tree structure can be defined by 3-tuples 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*

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

r is a name of the 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.

The class of computation trees is a subclass of $Tree$, which contains mappings from node to modules, and from nodes to its inherited/synthesized attributes, respectively.

Definition 4.4 *The class $CTree$ of computation tree is defined by*

$$\begin{array}{rcl} CTree & \stackrel{def}{=} & (T, M, I, S) \\ T = (r, N, C) & : & Tree \\ M & : & Node \to Module \\ I, S & : & Node \to \mathcal{P}(Atr) \end{array}$$

Atr and $Module$ are the class of attribute and module, respectively, and are treated as primitives. 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.

The progress of a computation is represented by the process of transforming computation trees using decomposition and attribute calculation. Each step of the transformation can be described formally as a transform function $Exec : CTree \times Node \times D \to CTree$, where D denotes the class of decompositions shown in Definition 4.2

$Exec$ is defined as follows:

Definition 4.5 *For $CT = (T, M, I, S) : CTree, n_0 \in N, \forall i C(n_0, i) = \perp$,*

$$\begin{array}{l} Exec(CT, n_0, \{return\ \textbf{where}\ E\}) = Calculate(CT, n_0, E) \\ Exec(CT, n_0, \{M_1, \ldots, M_k\ \textbf{where}\ E\}) = Calculate(Grow(CT, n_0, \{M_1, \ldots, M_k\}), n_0, E) \\ Exec(CT, n_0, \{\textbf{for}\ R\ \textbf{do}\ M\ \textbf{where}\ E\}) = \\ \quad Calculate{-}iter(Grow{-}iter(CT, n_0, count(R), M), n_0, index(R), E) \end{array}$$

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 $Grow$ or $Grow{-}iter$, with attributes being calculated by $Calculate$ or $Calculate{-}iter$ depending on the structure of decomposition. For details of these functions, see [17].

Execution Sequence. The program is executed by repeated applications of $Exec$, starting with an initial tree containing only a root node. Figure 12 illustrates this process; $Grow$ adds new nodes to the tree, while $Calculate$ determines the values associated with each node.

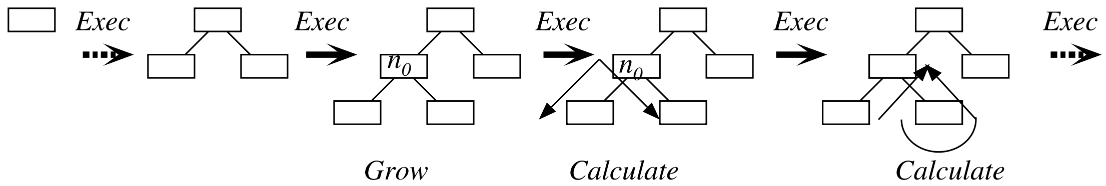

Figure 12: Execution by Series of Tree Transformations

4.3 Semantics of Fault-Tolerance Features

The largest difference when considering fault-tolerance in FTAG is in the growth of a computation tree. In the standard case when only normal or iterative decompositions are performed, computation trees grow monotonically. However, this is no longer true when considering features such as redoing, which alter the normal pattern of execution. To deal with this, we augment the formal semantics to support such non-standard execution.

Consider the case of redoing. The actual writing of the checkpoint of inherited attributes has no effect on the computation since all that must be done is to write the current values to the stable object base. However, once the computation executes a redoing decomposition that requires rolling back to the checkpoint, the fragment of the computation tree following the checkpoint must be cut from the computation tree.

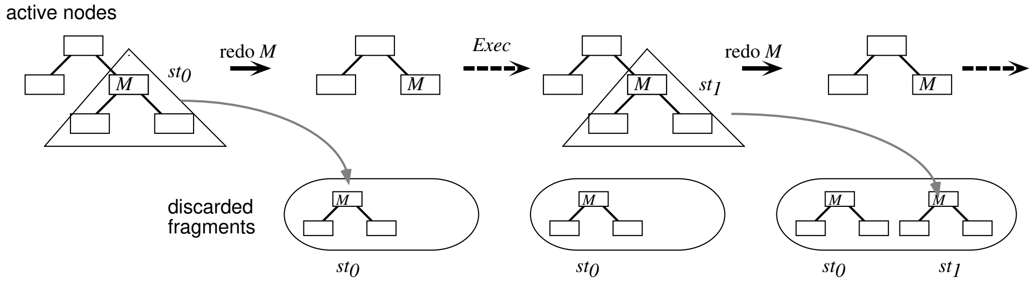

Figure 13: Tree Transformation in Redoing

Figure 13 illustrates the tree transformation process for redoing. The fragment st_0 is first removed and replaced by a new node representing the new computation when a redo M is encountered. The following computation starts from the new node and forms a new fragment st_1. If another redoing occurs in st_1, it is cut from the tree in the same way as st_0.

We can describe this sequence formally by extending the definition of $Exec$.

Definition 4.6 *For* $CT = (T, M, I, S) : CTree, n_0 \in N, \forall i C(n_0, i) = \bot$,

$$Exec(CT, n_0, \{\ \textbf{redo}\ M\ \}) = Prune(CT, m)$$
$$where\ \ m\ is\ the\ most\ recent\ node\ corresponding\ to\ M.$$

Function $Prune$ returns a tree eliminating all nodes under m except m itself.

Successive checkpoints extend this notion in the natural way. When the computation reaches a checkpoint, previous checkpoints that have the same name but have been instantiated from a different node in the tree are overridden. This means that they cannot be restored as a result of a redoing operation unless the later checkpoints are deactivated explicitly. Figure 14 illustrates the tree transformations that occur in the case of successive checkpoints. When the computation reaches M, st_{00} is regarded as a current fragment. If it reaches another M, the trees $st_{10}, \ldots, st_{1k}$ are replaced, but not st_{00}. If a failure is later detected in st_{00}, the program should deactivate the second checkpoint using an erase operation. Once this second checkpoint is cleared, further redoings can be performed in which the first checkpoint is the target.

Replicated decomposition can be expressed in the same way as normal decomposition, except that a means of selecting a single return value from the replicas is needed. This feature does not affect the semantics, but rather the method for managing objects. The architecture of the stable object base is discussed below.

5 A Software Architecture for FTAG

In this section, we discuss an architecture for implementing the FTAG computation model in a computer system consisting of multiple processors and (logically) shared external storage (e.g., disk). Specifically, we

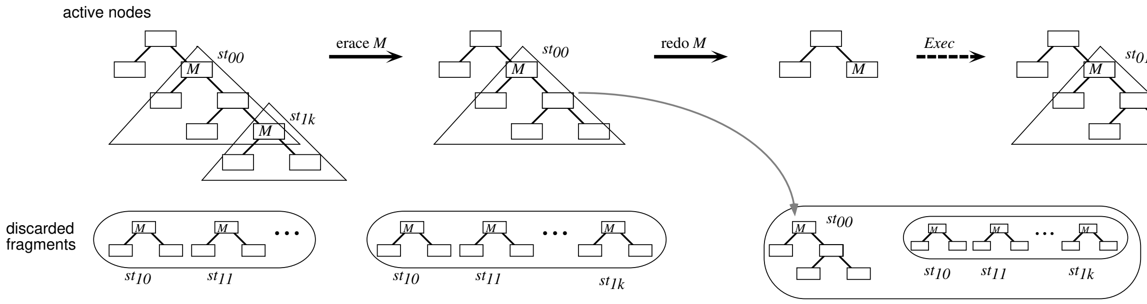

Figure 14: Tree Transformation for Successive Checkpoints

address the issues of the use of external storage to implement a stable object base for storing attribute values, and processors and module allocation.

5.1 Workspaces

As described in Section 2.2, the FTAG model includes a stable object base supporting versioned objects to which attribute values can be written. Here, we consider how such an object base might be implemented using disk storage, with a special focus on the versioning aspect.

Versioned objects are required for realizing the checkpoint mechanism needed for the redoing mechanism. Such objects must have the following properties:

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.

In order to realize these properties, we define a structure called a $workspace$ as the part of stable storage in which *vital attributes* are stored, i.e., the minimum sets of attributes needed for calculating subsequent values following a redoing operation.

First, we define the sets of modules $checkpoint\ modules$ and $replicated\ modules$ as follows:

Definition 5.1 *The set of checkpoint modules $\mathcal{C}$ contains all modules M that are denoted as checkpoints, i.e. modules that appear in $\{$**redo** $M\}$ in any part of the program.*

Definition 5.2 *The set of replicated modules $\mathcal{R}$ contains all modules M that are replicated, i.e., modules that are decomposed into submodules that are identical in the numbers and types of inherited and synthesized attributes.*

Note that both types of modules can be determined statically from the program text.

Given this, vital attributes are defined as all inherited attributes of checkpoint modules, and all inherited and synthesized attributes of replicated modules. Synthesized attributes of replicated modules are included since they must be available for any selection mechanism that reduces the multiple return values into a single one.

These features can be described formally by extending the definition of $CTree$ to include a mapping from attribute names to the storage where an attribute value is actually stored.

Definition 5.3 *The class $CTree$ of computation tree is defined by*

$$
\begin{array}{rcl}
CTree & \stackrel{def}{=} & (T, M, I, S, \mathcal{V}) \\
T = (r, N, C) & : & Tree \\
M & : & Node \to Module \\
I, S & : & Node \to \mathcal{P}(Atr) \\
\mathcal{V} & : & Atr \to WS
\end{array}
$$

$\mathcal{V}$ is a mapping from attributes to workspaces in which the values are stored. If $\mathcal{V}(a) = \perp$, a is mapped to volatile storage such as main memory. The notation $domain(\mathcal{V})$ is used to refer to the set of attributes stored in any workspace. Hence, the addition of an attribute to $domain(\mathcal{V})$ implies the allocation of new workspace for that attribute, while its deletion means destruction of the value and release of the workspace.

Rules for changing $\mathcal{V}$ when the computation reaches m are described as follows:

(Rule 1) If $m \notin \mathcal{C} \cup \mathcal{R}$, no workspace is created and $\mathcal{V}$ is unchanged.

(Rule 2) If $m \in \mathcal{C}$, a new workspace $ws : WS$ is created and all $a \in Inh(m)$ are stored in ws, then $\mathcal{V}' = \mathcal{V} \cup \{(a, ws) \mid a \in Inh(m)\}$

Figure 15 shows a generic form of checkpoint use, while Figure 16 illustrates the change in the computation tree. In this example, $\mathcal{C} = \{M_2\}$. The first time the computation reaches M_2, a workspace ws is created and u is stored as the initial version u_0 (Step (a)). Computations under M_2 are performed using this workspace until another checkpoint is needed. All computations redone from M_2 only need the values of the inherited attribute of M_2, so redoing can be realized using this mechanism. When another M_2 appears under the subtree of the current module, another workspace WS_1 is created and u is stored (Step (b)).

$$
\begin{array}{ll}
M(x \mid y) & \Rightarrow M_1(x \mid v)\;\; M_2(v \mid y) \\
M_2(v \mid y) & \Rightarrow M_{21}(v \mid u)\;\; M_{22}(u \mid y) \\
M_{22}(u \mid y) & \Rightarrow [\;\; u = \perp \quad \to \textbf{redo}\; M_2 \\
 & \quad\;\; \mid \textbf{otherwise} \to M_{22}body(u \mid y)\;\;] \\
M_{22}body(u \mid y) & \Rightarrow \;\; \ldots \;\; M_2(u \mid w) \;\; \ldots
\end{array}
$$

Figure 15: Example of using checkpoints

As noted above, checkpoints may appear in succession. Assume that M is also a checkpoint in Figure 16, i.e. $\mathcal{C} = \{M, M_2\}$. Then, these two modules form successive checkpoints and ws_0 through ws_3 are created, with each holding the value of x or u at that instant. Versioned objects V_x and V_u are realized as shown in Figure 17.

In the case of replicated decomposition, workspaces are created in the same way. This can be described using the following rules:

(Rule 3) If $m \in \mathcal{R}$, a new workspace ws_0 is created and both $a \in Inh(m)$ and $a \in Syn(m)$ are stored in ws_0; workspaces ws_i, corresponding to each replica m_i, are then created.
$$
\mathcal{V}' = \mathcal{V} \cup \{(a, ws_0) \mid \forall a \in Inh(m) \cup Syn(m)\} \;\cup \\
\qquad\qquad\qquad \{\; (a_i, ws_i) \mid \forall a_i \in Inh(m_i), 1 \leq i \leq k \;\}
$$

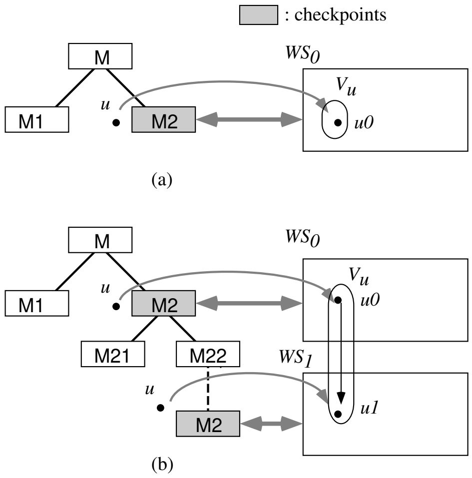

Figure 16: Workspace for single checkpoints

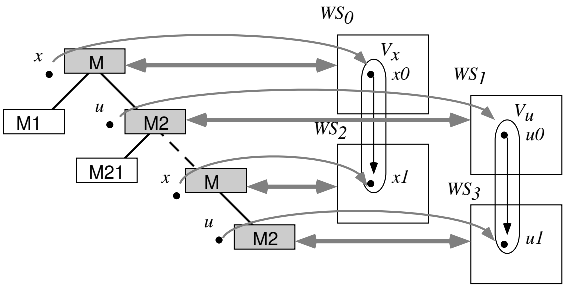

Figure 17: Workspace for successive checkpoints

Notice that the change of $\mathcal{V}$ describes only the new storage allocation for the attribute without calculating its value. Consider the following example:

$$M(x \mid y) \quad \Rightarrow M_1(x \mid y1) \;\; M_1(x \mid y2) \;\; M_1(x \mid y3)$$
$$\textbf{where} \;\; y = vote(y1, y2, y3)$$

Figure 18 illustrates the relationship among workspaces used in this program. First, ws_0 is created corresponding to M, followed by the creation of workspaces for each replica ws_1 through ws_3. The value of x in each replica is copied from x in ws, then dependencies from x to x_i in V_x are established.

5.2 Processors and Module Allocation

Since execution in FTAG depends only on having the appropriate attributes present, a simple scheme can be used for allocating decompositions to processors. In particular, a node in the computation tree is assigned to a processor upon creation, with that processor being responsible for all communication of attribute values between the node and its children.

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,

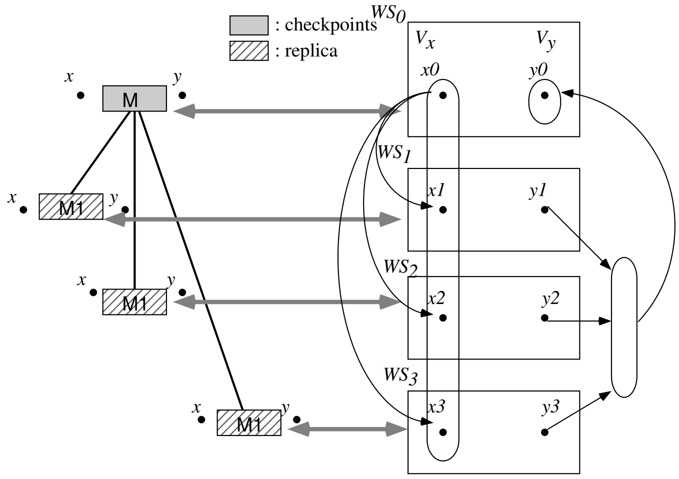

Figure 18: Workspaces for replicated modules

$$Proc = \mathcal{P}(proc)$$
$$proc = (Send, Recv)\ where\ Send, Recv : \mathcal{P}(Atr)$$

$Proc$ 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. The assignments of nodes to processors are described formally by a total function that maps a processor to the set of nodes assigned to that processor, $Ass : proc \to \mathcal{P}(Node)$.

In order to minimize the cost for communication between a processor and secondary storage, we assume that software and hardware failures affect only the processor on which the failure occurs. With this assumption, execution cost can be reduced by storing values of vital attributes at other processors rather than on disks. That is, the stability of a workspace is approximated by having its values stored in the memory of multiple processors, so that they remain available as long as at least one such processor is functioning [5]. The probability of losing a value can then be reduced to something arbitarily close to zero by storing enough copies. Below, we assume that a workspace is implemented by storing two copies, one of which is the processor executing a module.

The rules for modifying Ass when m is created are as follows:

(Rule 1) If $m \notin \mathcal{C} \cup \mathcal{R}$, m is assigned to the same processor as its parent(say p).
$Ass' = Ass \cup \{(p, m)\}$

(Rule 2) If $m \in \mathcal{C}$, m is allocated new processor p_0 and another copy p'_0.
$Ass' = Ass \cup \{(p_0, m), (p'_0, m)\}$

(Rule 3) If $m \in \mathcal{R}$, it is considered a kind of checkpoint then m is allocated p_0 and p'_0 , and each replica $m_1, \ldots, m_k$ are allocated the different processor $p_1, \ldots, p_k$.
$Ass' = Ass \cup \{\ (p_0, m), (p'_0, m)\ \}\ \cup\ \{\ (p_i, m_i) \mid 1 \le i \le k\ \}$

In (Rule 2), p'_0 is the extra processor used for emulating the stable workspace, so only p_0 is involved in the regular computation. Should redoing from m occur, m and its descendants in p_0 are discarded and the original m is copied from p'_0.

Attribute values are passed from one sender to one receiver in the case of normal decomposition. On the other hand, a mechanism such as multicast for sending values to more than one receiver, and a multi-way receive for receiving values from many senders, are required for realizing communication in replicated decompositions. In this case, for example, p_0 multicasts the inherited attribute x to processors $p_1, \ldots, p_k$. Upon receipt, each processor starts its computation using the value.

This type of multicast mechanism is useful for managing both fail-stop and Byzantine failures[16]. In the case of fail-stop, all processors that generate results are considered to be working correctly. Thus, when one, say p_1, finishes its computation, it sends the result y, which is used immediately. For Byzantine failures, the value of y must be collected from each replica, with the return value being calculated by the collation function. Although the specific multicast mechanism may have to vary depending on the type of failure to be tolerated, from the perspective of the rest of the implementation, the only difference is that a selection mechanism has to be used in the Byzantine case.

Finally, the port mechanism can also be used to implement the requirement that $\perp$ be assigned to attributes of modules executing on processors that have failed. Specifically, should no message from a particular processor be received within a specified amount of time, it is assumed to have failed and $\perp$ is assigned to the appropriate attributes. This technique is, of course, just the standard use of timeouts for failure detection.

6 Discussion

Advantages. The functional and attribute-based approach to implementing fault-tolerant software embodied in FTAG has several advantages over traditional imperative approaches. For example, it exhibits high degrees of referential transparency and composability, which makes the description easier to read and understand. One reason for this is the locality of the notation; information is only passed between functions using attributes, and then only between functions that have a parent/child relationship. As a result, it is easy to determine attribute dependencies, which facilitates parallel execution. Moreover, in FTAG, module decomposition only represents the structure of the computation and relationships among the attributes, rather than the way the computation is performed. Since this means that the computation order is determined implicitly by attribute dependencies, it is easier to understand the effect of the computation.

The functional aspect of FTAG also makes the program easier to analyze and provides opportunity for optimizations. Consider the following example:

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

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 it is 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 data-flow analysis of the program could expose this optimization in the implicit case as well, but only at significant cost.

Finally, FTAG is easy to implement using a multicast communication mechanism as described in section 5.2. The nature of the computation makes it possible to describe the processor allocation strategy outside of the program script, leaving ample opportunity for optimizing the strategy based on the specifics of the computation.

Related work. Other researchers have also explored functional approaches to implementing fault tolerance in various situations. In [13], a functional model is proposed as a means for implicitly implementing graceful degradation when processors fail in the custom Fault Tolerant Parallel Processor (FTPP) developed at Draper Labs. A program written using this model is composed of Remote Procedures (RPs), which are functional modules that are allocated to the various processors during execution. Although functional in their execution, RPs can be written in any standard language such as C or Ada. Should a processor fail while executing an RP, that module is reassigned to another processor transparently to the application and reexecuted from the beginning with its original arguments. The functional nature of the module ensures that the results will be the same as the original execution.

A functional approach for implicit application-transparent fault tolerance is presented in [14]. The emphasis in this work is on dealing with operational faults using an intensional model in which an implicit context is associated with each value computed by the program. This model supports a demand-driven implementation in which a value is only computed when necessary, which makes it easy to realize fault tolerance by reissuing a request for a value should the first attempt fail. Again, the functional nature of the execution guarantees that the result will be the same. The possibility of incorrect values can be dealt with by issuing the value request multiple times and then using a collation function on the results.

The advantages of referential transparency and composability for fault-tolerant parallel computing are also discussed in [11]. In that paper, a functional language and dataflow computing model for designing large-scale parallel computations are described. Redundancy schemes such as recovery blocks and N-version programming are then used as illustrative examples of how fault tolerance can be programmed using this approach.

While FTAG is similar to the these approaches in its exploitation of functional programming for fault tolerance, it also differs in a number of respects. For example, FTAG supports the handling of both software and operational failures within a single framework, whereas each of the others only handles one or the other. Also, when compared with [11], FTAG promotes a more dynamic approach to failure handling since recovery procedures can be created incrementally at runtime, rather than being constrained to a static approach. In addition, unlike the other approaches, FTAG derives benefits from being based on attribute grammars, as well as from the functional aspect of the approach.

Another difference is that both [13] and [14] provide fault tolerance implicitly—essentially, every module is treated as a checkpoint module using our terminology—whereas FTAG provides mechanisms such as redoing and replicated decomposition that the programmer can use to implement common fault-tolerance paradigms easily. The former has the advantage of transparency to the application, while tha latter provides more flexibility and user control. Also note that it would easy to implement an implicit approach similar to the others in FTAG by making every module a checkpoint module and using a system wrapper around modules to detect certain types of failures and execute the appropriate redo operation.

7 Conclusions

The FTAG computational model described in this paper provides a unified framework that supports development of programs that can tolerate software and/or operational faults. This is achieved by augmenting a functional approach to programming based on attribute grammars with support for common fault-tolerance paradigms such as recovery and replication. The specific mechanisms—redoing and replicated decomposition—maintain the linguistic integrity of the notation, and are easily implemented using a stable object base with versioning and multicast communication. The ability to infer automatically those values that must be stored in stable storage for recovery purposes is also a significant advantage. In addition to outlining the approach and examples, we presented a formal description of FTAG, which specifies precisely the semantics of the approach and how the fault-tolerance features are integrated with normal processing.

As noted in the Introduction, this model is derived from similar attribute based models that have proven useful is other areas, including object-oriented programming and the description of high-level software design processes. The experience with the software design process is especially significant, 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. We expect that further exploration of the similarities between the two areas will highlight additional connections and provide additional possibilities for transitioning techniques.

Other aspects of future work will concentrate in two different areas. One is implementing a programming system based on the software architecture described in section 5 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 characterizing other fault-tolerance paradigms using this framework, and investigating the features needed to realize each paradigm. For example, multicast was used above to simplify the implementation of replication both for fail-stop and Byzantine failures. We expect to identify similar abstractions that would be useful for other paradigms. In both cases, our efforts will include investigating realistic applications to test the true benefits of this approach.

References

[1] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 220–232, Jun 1975.

[2] A. Avizienis, "The N-Version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1491–1501, 1985.

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

[4] B. Liskov, "The Argus language and system," in *Distributed Systems: Methods and Tools for Specification, Lecture Notes in Computer Science, Volume 190* (M. Paul and H. Siegert, eds.), ch. 7, pp. 343–430, Berlin: Springer-Verlag, 1985.

[5] F. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, pp. 299–319, Dec 1990.

[6] D. Knuth, "Semantics of context-free languages," *Mathematical Systems Theory*, vol. 2, no. 2, pp. 127–145, 1968.

[7] T. Katayama, "HFP, a hierarchical and functional programming based on attribute grammars," in *Proceedings of the 5th Int. Conference on Software Engineering*, pp. 343–353, 1981.

[8] Y. Shinoda and T. Katayama, "Attribute grammar based programming and its environment," in *Proceedings of the 21st Hawaii Int. Conf. on System Sciences*, (Kailu-Kona), pp. 612–620, Jan 1988.

[9] 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 Int. Workshop on Attribute Grammar and its Application*, vol. 461, pp. 177–191, LNCS Springer-Verlag, 1990.

[10] T. Katayama, "A hierarchical and functional software process description and its enaction," in *Proceedings of the 11th Int. Conf. on Software Engineering*, pp. 343–352, 1989.

[11] A. Bondavalli and L. Simoncini, “Functional paradigm for designing dependable large-scale parallel computing systems,” in *Proceedings of the Distributed Computing Systems*, pp. 108–114, 1993.

[12] D. Grit, “Towards fault tolerance in a distributed applicative multiprocessor,” in *Proceedings of the 14th Symposium on Fault-Tolerant Computing*, (Orlando, FL), pp. 272–277, June 1984.

[13] R. Harper, , G. Nagle, and M. Serrano, “Use of a functional programming model for fault tolerant parallel programming,” in *Proceedings of the 19th Symposium on Fault-Tolerant Computing*, (Chicago, IL), pp. 20–26, Jun 1989.

[14] R. Jagannathan and E. Ashcroft, “Fault tolerance in parallel implementations of functional languages,” in *Proceedings of the 21st Symposium on Fault Tolerant Computing*, (Montreal, Canada), pp. 256–263, Jun 1991.

[15] M. Suzuki, A. Iwai, and T. Katayama, “A formal model of re-exection in software process,” in *Proceedings of the 2nd Int. Conf. on Software Process*, (Berlin, Germany), pp. 84–99, Feb 1993.

[16] M. Suzuki, T. Katayama, and R. D. Schlichting, “Implementing fault-tolerance with an attribute and functional based model,” in *Proceedings of the 24th Symposium on Fault-Tolerant Computing*, (Austin, TX), pp. 244–253, Jun 1994.

[17] M. Suzuki, T. Katayama, and R. D. Schlichting, “A functional and attribute based computational model for fault-tolerant software,” Tech. Rep. TR 93-8, Dept of Computer Science, University of Arizona, Tucson, AZ, 1993.

[18] B. Lampson, “Atomic transactions,” in *Distributed Systems—Architecture and Implementation*, pp. 246–265, Berlin: Springer-Verlag, 1981.

Appendix: FTAG Grammar

$$
\begin{array}{lll}
<\mathit{programs}> & ::= & <\mathit{type\_defs}>\ <\mathit{module\_defs}> \\
<\mathit{type\_defs}> & ::= & <\mathit{typedef}>\ [\ <\mathit{typedef}>\ ]* \\
<\mathit{typedef}> & ::= & \mathbf{type}\ <\mathit{type\_name}>[\,,<\mathit{type\_name}>]* \\
 & & \ =<\mathit{type\_denoter}> \\
<\mathit{type\_denoter}> & ::= & <\mathit{primitive\_types}>\ \mid\ <\mathit{type\_constructor}> \\
<\mathit{type\_name}> & ::= & \mathbf{id} \\
<\mathit{primitive\_types}> & ::= & \mathit{int}\mid \mathit{char}\mid \mathit{real}\mid \mathit{string}\mid ... \\
<\mathit{type\_constructor}> & ::= & \mathbf{array}\ <\mathit{index}>\ \mathbf{of}\ <\mathit{type}>\ \mid ... \\
<\mathit{index}> & ::= & [' <\mathit{range}>[,<\mathit{range}>]*\ ']' \\
<\mathit{range}> & ::= & <\mathit{number}>..<\mathit{number}> \\
<\mathit{module\_defs}> & ::= & <\mathit{module}>\ [\ <\mathit{module}>\ ]* \\
<\mathit{module}> & ::= & <\mathit{module\_name}>\ (\ <\mathit{attr\_list}>\ '|'\ <\mathit{attr\_list}>\ ) \\
 & & \quad \Rightarrow\ <\mathit{decomposition}> \\
<\mathit{module\_name}> & ::= & \mathbf{id} \\
<\mathit{attr\_list}> & ::= & <\mathit{attr}>[,<\mathit{attr}>]* \\
<\mathit{attr}> & ::= & <\mathit{type}>\ \mid\ <\mathit{type}>.<\mathit{modifier}> \\
 & & \mid\ '*'<\mathit{type}>.<\mathit{modifier}> \\
<\mathit{modifier}> & ::= & \mathbf{id} \\
<\mathit{decomposition}> & ::= & <\mathit{unconditional\_decompostion}> \\
 & & \mid\ <\mathit{conditional\_decompostion}> \\
<\mathit{unconditional\_decomposition}> & ::= & <\mathit{primitive}> \\
 & & \mid\ <\mathit{normal\_decompostion}> \\
 & & \mid\ <\mathit{iterative\_decompostion}> \\
 & & \mid\ <\mathit{redoing\_decomposition}> \\
<\mathit{primitive}> & ::= & \quad \mathit{return}\ \mathbf{where}\ <\mathit{attr\_relations}> \\
<\mathit{normal\_decomposition}> & ::= & \mid\ <\mathit{submodules}>\ \mathbf{where}\ <\mathit{attr\_relations}> \\
<\mathit{submodules}> & ::= & <\mathit{submodule}>[<\mathit{submodule}>]* \\
<\mathit{submodule}> & ::= & <\mathit{module\_name}> \\
 & & \ (\ <\mathit{attr\_list}>\ '|'\ <\mathit{attr\_list}>\ ) \\
 & & \mid\ '\{'\ <\mathit{submodule}>\ [\ ;\ <\mathit{submodule}>\ ]*\ '\}' \\
<\mathit{attr\_relations}> & ::= & <\mathit{attr\_exp}>\ [\ <\mathit{attr\_exp}>\ ]* \\
<\mathit{attr\_exp}> & ::= & <\mathit{attr}>\ =\ <\mathit{function}>\ (\ <\mathit{attr\_list}>\ ) \\
<\mathit{iterative\_decompostion}> & ::= & \mid\ \mathbf{for}\ <\mathit{attr}>=<\mathit{range}>\ \mathbf{do}\ <\mathit{submodules}> \\
 & & \quad \mathbf{where}\ <\mathit{attr\_relations}> \\
<\mathit{conditional\_decompostion}> & ::= & '['\ <\mathit{condition}>\ \rightarrow\ <\mathit{decomposition}> \\
 & & \ [\ '|'\ <\mathit{condition}>\ \rightarrow\ <\mathit{decompostion}>\ \ ]* \\
 & & \quad '|'\ \mathbf{otherwise}\ \rightarrow\ <\mathit{decomposition}>\qquad ']' \\
<\mathit{condition}> & ::= & <\mathit{boolean\_expression}> \\
<\mathit{redoing\_decomposition}> & ::= & \mathbf{redo}\ <\mathit{module\_name}>
\end{array}
$$