Abstract

Computational modeling tools are critical to engineering. In the absence of a sufficiently complete, mathematically precise, abstract specification of the semantics of the modeling framework supported by such a tool, rigorous validation of the framework and of models built using it is impossible; there is no sound basis for program implementation, verification or documentation; the scientific foundation of the framework remains weak; and significant conceptual errors in framework definition and implementation are likely. Yet such specifications are rarely defined. We present an approach based on the use of formal specification and denotational semantics techniques from software engineering and programming language design. To illustrate the approach, we present elements of a formal semantics for a dynamic fault tree framework that promises to aid reliability analysis. No such specification of the meaning of dynamic fault trees has been defined previously. The approach revealed important shortcomings in the previous, informal definitions of the framework, and thus led to significant improvements, suggesting that formally specifying framework semantics is critical to effective framework design.

1. Introduction

Computational analysis of system models is fundamental to engineering. A model represents a system in some modeling framework that comprises a set of modeling constructs and composition mechanisms. A model is analyzed to produce information about the modeled system. For example, fault tree models [10] are often analyzed to estimate system reliability.

A serious risk created by our growing dependence on modeling tools is that engineers will make adverse decisions based on computed results that are not valid. If the cost of such a decision is potentially large and if there is no sound basis for assessing validity, then it is uninformed and even unethical for an engineer to accept such results.

For the results produced by such an analysis to be validated, four conditions must be satisfied. First, the meaning of the framework, and thus of models expressed in it, must be specified: completely in all areas of uncertainty, abstractly, and with mathematical precision. Second, the specification must be validated by domain experts. Third, the software implementing the framework must be verified against the specification. Finally, any model in the framework must be validated against the system modeled. The first condition is the most critical, because framework validation, program verification and model validation all require a precise definition of framework semantics.

Techniques for specifying complex logical structures have been developed, mainly in the software engineering and languages (SEL) research community. However, they are rarely applied, either in industry or by non-SEL researchers who design modeling frameworks. SEL research has shown that in the absence of such specification, important conceptual errors in design are generally overlooked. As a consequence, many modeling frameworks and tool implementations are not suitable for use in critical engineering contexts. Many engineers are unaware of this difficulty and the risks that it presents; nor do SEL researchers fully understand how to disseminate the techniques that have been developed for rigorous specification.

We present results from a multi-disciplinary collaboration between software engineering researchers and researchers who are designing a new modeling framework. The collaboration involves the use of formal software specification to define the semantics of a framework for the reliability analysis of computer-based, fault-tolerant systems. In essence it is a traditional fault tree framework [10] extended with new constructs for expressing order-dependent
failures, shared pools of spares, imperfect coverage, and common-cause failures [5]. There are two primary contributions of this work. First, we are defining a precise semantics for an important modeling framework, establishing it on a firmer scientific foundation. Second, our case study suggests that multi-disciplinary collaborations between specification and domain experts can significantly help to overcome impediments to the use of formal specification techniques.

The rest of this paper is organized as follows. Section 2 introduces the dynamic fault tree modeling framework informally. Section 3 describes dynamic fault trees informally and some of complexities in defining their semantics. Section 4 provides an overview of the specification, Section 5 provides an introduction to the Z specification language, and Section 6 presents an abstraction of real numbers. The next five sections (7, 8, 9, 10, and 11) describe the key components of the specification, and how the specification addresses the complexities described in Section 3. Section 12 discusses related work. Section 13 summarizes and describes future work.

2. The Dynamic Fault Tree Framework

The traditional static fault tree modeling framework [10] allows one to model how boolean combinations of component-level failure events produce system failures. Fault trees are useful in part because they have intuitive graphical depictions. Figure 1 depicts a static fault tree. The interior nodes are called gates. They represent failure events that occur if their input events occur in certain combinations. The gate Physical Damage to the Core is an OR gate. The event corresponding to this gate occurs if the Mechanical Damage or the Explosive Damage event occurs. The other essential gate type in static fault trees is AND. The corresponding event occurs if all input events occur. The leaves of a tree are called basic events. Their failure characteristics are modeled by probability distributions and failure coverage models. The top-most event represents system-level failure. Given the tree and the basic event parameters, a system level probability of failure is computed.

Dynamic fault tree (DFT) modeling frameworks augment static fault trees with constructs that are important for modeling fault-tolerant systems with complex redundancy management [3, 5]. Most fundamentally, DFTs include constructs for modeling how sequences of failure events cause system failures. Other constructs model imperfect coverage, dynamic allocation of spares from pools, and common-cause failures (or functional dependencies).

The traditional approach to reliability analysis of such systems is based on Markov models. However, Markov models of complex systems are often unmanageable in size. DFTs are compact representations that in many cases can be mapped automatically to Markov models, relieving the analyst of the tedious and error-prone task of developing those models by hand. At a high level, then, some DFTs clearly have semantics that are expressible in terms of underlying Markov models.

However, no precise specification of DFT semantics of has been developed before. The problem is that the semantics turn out to be much more subtle and complex than they appear at first. The specifications to date [2, 3, 5, 6, 8] are inadequate to handle this complexity. Informal prose descriptions exist, but they are incomplete and inherently ambiguous. Mappings of DFTs to Markov chains have been defined for special cases, but they do not capture the general case. Source code and executable implementations exist and are precise, but procedural code is resistant to validation, and in the absence of a high-level specification there is no basis for rigorous program verification [7].

We present a precise, reasonably complete, abstract semantics for DFTs. We do so employing methods of formal specification and denotational semantics. We use the formal specification notation Z [9], which supports structured specifications based on predicate logic and typed set theory. From denotational semantics we adopt the general idea that one specifies semantics in terms of a mapping between two domains: a syntactic domain (DFTs), a semantic domain whose objects embody meanings, and a mapping that associates to each syntactic object a semantic object. The objects of our semantic domain are called failure automata (FA). They are closely related to Markov chains; and a semantics for such automata is defined in precisely the same style. Space limitations require that we we elide aspects of our specification. A complete specification is forthcoming as a technical report.


In this section we introduce DFT modeling constructs in more detail, but still informally. We also show how such an informal specification can leave important semantic issues unresolved. The main DFT modeling constructs are as follows.

![Figure 1. Example Fault Tree](image-url)
• **Replicated Basic Events:** Basic events model unelaborated events using probability distributions and coverage models. As a convenience to the user, basic events can have a *replication value*, which allows a basic event to represent several identical events connected to the same locations in the fault tree. This is particularly useful in conjunction with the *spare gate*, where replacement components can be taken from a pool of identical components until that pool of components is exhausted. Basic events also have a *dormancy factor*, which attenuates the failure rate when the basic event is used as a warm spare (see below).

• **AND:** The output event occurs if all the input events have occurred.

• **OR:** The output event occurs if any of the input events have occurred.

• **KOFM:** The output event occurs if at least $k$ out of $m$ of the input events have occurred.

• **Priority AND (PAND):** The output event occurs if all the input events have occurred and they occurred in the order in which they appear as inputs.

• **Cold, Warm, Hot Spare Gates (CSP, WSP, HSP):** When the primary input fails, available spare inputs are used in order until none are left, at which time the gate fails. Spares can be shared among spare gates, in which case the first spare gate to utilize the spare makes the spare inaccessible to the other spare gates. The “temperature” of a spare gate indicates whether unused spares can not fail (cold), fail at a rate attenuated by the dormancy factor of the spare (warm), or fail at their full rates (hot).

• **Sequence Enforcer (SEQ):** Asserts that events can occur only in a given order. This is not a gate in the sense that it has no output event.

• **Functional Dependency (FDEP):** Asserts a functional dependency—that the failure of the trigger event causes the immediate and simultaneous failure of the dependent basic events. This is not a gate in the sense that it has no output event.

Informal definitions such as the ones above are likely to contain errors. In fact, the above definitions do have errors. Consider the fault tree in Figure 2. Here, the FDEP gate models a system in which the occurrence of Event A causes Event B and Event C to occur. The event associated with the PAND gate occurs if Event B and Event C occur in order. The questions concern the meaning of the FDEP and the PAND. Do Event B and Event C occur simultaneously? Do they occur simultaneously with, or after, Event A? If simultaneously with each other, will the PAND event occur, or does it occur only if Event B occurs before Event C? Such questions must be answerable on the basis of a precise, abstract specification if such a DFT is to have a well defined meaning.

These questions actually arose from a model constructed by an engineer at Lockheed-Martin in our DFT framework. There was no specification, and so no way for the engineer to validate his model. He could not even infer a specification from the tool behavior because the tool did not treat simultaneity consistently in all cases. The reason is that the implementor had no specification to meet, and thus was at liberty to make decisions about the framework semantics.

![Figure 2. A subtlety concerning simultaneity](image1)

A key point is that it is inadequate to specify the meanings of individual modeling constructs in isolation—whether informally or not. What is needed is the definition of the meaning of an arbitrary DFT in which these constructs might interact in subtle ways. The next example illustrates this point. Consider the portion of a fault tree depicted in Figure 3. The spare gates, Spare Gate 1 and Spare Gate 2, are using Event B and Event C, respectively, as indicated by heavy lines. They also share a spare event, Event D, that is not currently in use. The FDEP indicates that if Event A occurs, Event B and Event C occur simultaneously with it. In this case, Spare Gate 1 and Spare Gate 2 have to contend for the single shared spare, Event D. There are two possible outcomes. Developing the specification showed that next-state relation is non-deterministic.

![Figure 3. A subtlety concerning non-determinism](image2)

The non-determinism in this example was recognized only when we specified the framework with mathematical
precision. Even recognizing such conceptual difficulties appears to require the application of formal specification techniques. Our efforts revealed many such subtle issues, which we now enumerate. In presenting our formal specification in the following sections, we will refer back to some of these issues, showing how we identified and resolved some of them.

- **Issue #1**: Replication, although seemingly innocuous, causes problems because basic event replicates are anonymous with respect to order. If a replicated basic event is connected to a PAND gate, for example, it is not clear whether the failure of a replicate should be considered to be in-order or out-of-order.

- **Issue #2**: Two spare gates can experience contention for a single shared spare if both their currently used components are failed simultaneously by a functional dependency.

- **Issue #3**: Cold, warm, and hot spare gates not only model sparing behavior, but also affect the failure rates of basic events attached to them. As a result, basic events can not be connected to spare gates of different types, because the attenuation of the failure rate would not be defined.

- **Issue #4**: Fault trees must maintain a history of event occurrences, because the specification of gates such as the PAND depend on the past order of event occurrences.

- **Issue #5**: A dependent input to an FDEP could be the trigger to another FDEP, which means that the occurrence of a single event can cause a cascading series of events to occur.

- **Issue #6**: Functional dependency gates raise simultaneity issues relevant to the semantics of gates sensitive to the order of event occurrences.

- **Issue #7**: FDEPs can be cyclicly dependent, which means that any basic event in the cycle can cause all the others to occur. This means that more than one basic event can result in the same transformation of a fault tree from one state to another.

4. Specification Strategy

The rest of this paper presents our specification. In this section, we introduce our approach to specification and the overall structure of our specification. Subsequent sections delve deeper into selected details.

The overall structure, which is based on ideas from denotational semantics, is depicted in Figure 4. First, we define a syntactic domain of DFTs. Second, we define a semantic domain whose objects are failure automata. Third, we formalize the semantics of DFTs by specifying a mapping from the DFT domain to the FA domain. We then apply this approach to FAs by mapping them to other semantic domains, such as Markov chains and models based on Boolean decision diagrams (BDDs). This specification approach can be repeated until mathematically well defined domains, such as Markov chains, are reached. We partition DFTs and FAs into several types, for which such mappings can be defined. In this paper we address the subset of DFTs whose semantics can be defined in terms of Markov chains, namely DFTs whose basic event probability distributions are Weibull or exponential only (as indicated by the dark lines in the figure).

![Figure 4. Specification strategy](image-url)
Figure 5. Example Failure Automaton

level failure states.

We define DFT semantics in terms of an intermediate representation, the FA. Like a Markov chain, an FA is a state-transition diagram. It is not a Markov chain, although in many cases it is isomorphic to one. FAs support the specification of semantics for certain DFTs for which there are no corresponding Markov chains, such as DFTs with constant probability distributions, which can be mapped to BDDs instead.

Figure 5 illustrates a portion of the FA for a simple DFT. In our current formulation, each FA state represents a fault tree state. A state of a fault tree represents which events have occurred, in what order (the event history), which spares are in use by which spare gates, and other information discussed in more detail below. We represent event histories in FA states because the order in which events occur is important in evaluating dynamic constructs, such as priority-and gates, and is thus critical in computing the next-state relation for an FA. In the figure, events that have occurred are in gray. The heavy line indicates which input to the spare gate is in use. In the leftmost state, only event B has already occurred, and the history at the bottom of the state illustration thus reflects only that event.

An arc between two FA states indicates the basic event whose occurrence caused the state transition. The transition to the upper state in the figure, labeled with an “A”, indicates that the occurrence of basic event A caused the state change. In the resulting state, A is gray and the history is augmented accordingly. In addition, the spare gate is now using the basic event S1 (rather than A) because A is no longer available.

Figure 6 presents the Markov chain that corresponds to this portion of the FA. Each state in the Markov chain corresponds to state in the FA, and, in this case, transitions also correspond one to one. In general, an FA can have multiple arcs between two states. A Markov chain will have a single transition corresponding to such a set of arcs. The transition rates between states of the Markov chain correspond to the rate of occurrence of the triggering basic events.

Figure 6. Example Markov Chain

5. The Z Formal Specification Notation

In this section we present a brief overview of the Z (pronounced zed) formal specification notation [9], which we use for our specification. The notation supports the structuring and composition of complex expressions in first-order, typed set theory. We describe only the key concepts and notations used in this paper.

In Z, every value has exactly one type. A type is a set of elements that is disjoint from sets that define other types. Z has a number of primitive types, such as natural numbers (N) and integers (Z). The specifier can define a new type of objects without specifying any details of the objects using a given set in Z, which is denoted using square brackets (“[GivenSet]”). Z also provides several mechanisms for defining new types from existing ones. For example, if S is a type, then seq S is a type that comprises the set of all sequences of items of type S; ieseq S is the set of injective sequences (without repeated elements); F S comprises all finite sets of elements of type S. Instances of a type can be declared. For example, a statement such as mySeq : ieseq Z defines a state element named “mySeq” whose value is in the set of sequences of integers.
Sequences are simply partial functions (indicated by \( \to \)) from positive naturals \( \langle \mathbb{N}_1 \rangle \) to values of a particular type, where the domain of the functions range from 1 to some value \( n \) representing the number of items in the sequence. A sequence is a function, and an expression such as \( \text{mySeq}(2) \) denotes the application of the function to the value 2, representing the value of the second item in \( \text{mySeq} \). The value is undefined if there is no element 2. #\text{mySeq} denotes the length of the sequence.

\( \text{Z} \) has a rich collection of notations for defining relations over sets. Algebraically different kinds of relations are indicated by different types of arrows. For example, \( A \to B \) is a type comprising the set of functions from the set \( A \) to the set \( B \). Given \( f : A \to B \), i.e., \( f \) is some function whose actual domain is a subset of \( A \) and whose co-domain is a subset of \( B \), \( \text{ran} f \) denotes the range of \( f \), and \( \text{dom} f \) its co-domain. Cross-product types are denoted by the cross operator, \( \times \), applied to the constituent types. Given a value, \( \text{tuple} : A \times B \), the elements of \( \text{tuple} \) are denoted as \( \text{first}(\text{tuple}) \) and \( \text{second}(\text{tuple}) \).

\( \text{Z} \) provides a mechanism, called the schema, which supports the modular structuring of specifications of complex types. In a nutshell, a schema defines a type by specifying the state components of an element of the given type in terms of types that have already been defined, e.g., by given sets or other schemas; and by specifying invariant relations over these state components that are satisfied by all elements of the given type. Consider an example.

\[
\begin{align*}
\text{Example 1} & \\
\text{is} : & \mathbb{F}, \mathbb{Z} \\
\text{j} : & \mathbb{N} \\
1 & \in \text{is}
\end{align*}
\]

This schema defines a new type, \text{Example 1}. Items above the middle line declare state components. The schema says that every value of the type has the specified state components: a non-empty finite set of integers \text{is} and a natural number \text{j}. Invariant relations are stipulated in the predicate parts of schema, below the dividing line. Elements of the \text{Example 1} type are such that 1 is in \text{is}. An expression such as \text{ex} : \text{Example 1} states that \text{ex} is a value of type \text{Example 1}, whose name is \text{ex}. The state components of \text{ex} are denoted by \text{ex.is} and \text{ex.j}.

\( \text{Z} \) provides a schema calculus that allows schemas to be composed. In this paper we will make use of schema inclusion. Consider the schema below:

\[
\begin{align*}
\text{Example 2} & \\
\text{Example 1} & \\
\text{k} : & \mathbb{Z} \\
2 & \notin \text{is}
\end{align*}
\]

Schema inclusion means that the declarations of the included schema are aggregated textually with the declarations of the including schema. State components with the same name must have the same type and they are unified. Predicates of the included schema are conjoined with the predicates of the including schema. \text{Example 2} is thus exactly equivalent to the following schema:

\[
\begin{align*}
\text{Example 2Together} & \\
\text{se} : & \mathbb{F}, \mathbb{Z} \\
\text{j} : & \mathbb{N} \\
k : & \mathbb{Z} \\
1 & \in \text{is} \\
2 & \notin \text{is}
\end{align*}
\]

Sets can be constructed using set comprehension. The following set comprehension defines the set of all squares of even numbers:

\[
\{ e : \mathbb{Z} | e \text{ mod } 2 = 0 \cdot e^2 \}
\]

\( e \) is declared to be an integer such that the remainder after dividing by two is zero. The statement after “\( \cdot \)” defines the element for the constructed set. The set of squares of even numbers, as a type, can be named in the following way:

\( \text{SquaresOfEvens} \equiv \{ e : \mathbb{Z} | e \text{ mod } 2 = 0 \cdot e^2 \} \)

\( \text{Z} \) also supports the definition of axioms, which pertain globally to a specification. They are declared in \( \text{Z} \) in the following way:

\[
\begin{align*}
\text{factorial} : & \mathbb{N} \rightarrow \mathbb{N} \\
\text{factorial}(0) & = 1 \\
\forall i : & \mathbb{N} \bullet \\
\text{factorial}(i) & = i \cdot \text{factorial}(i-1)
\end{align*}
\]

Here \text{factorial} is defined as a recursive function from natural numbers to positive natural numbers. The base case is defined as a predicate on the factorial function, and the factorial function is defined for all non-zero naturals in the normal way.

6. Finessing Real Arithmetic

We now begin to present essential elements of our specification. Because \( \text{Z} \) was designed to specify discrete state systems, it has no support for continuous mathematics or real numbers. (The reasons are beyond the scope of this paper.) However, it is important for us to represent real numbers abstractly. Thus we begin by defining an abstract system of real numbers and operations in this section.

[8]
We introduce \( \mathbb{R} \) as a given type, and declare \( 0_\mathbb{R} \) and \( 1_\mathbb{R} \) to be elements of that type.

\[
\begin{align*}
0_\mathbb{R} & : \mathbb{R} \\
1_\mathbb{R} & : \mathbb{R}
\end{align*}
\]

We introduce functions that abstract the computation of the sum of a set of reals, as well as the sum, product and quotient of two reals, and real comparison. We declare the types of the functions, but do not constrain them to satisfy axioms of real or floating point arithmetic. We also introduce the function \( \text{intToReal} \), which is used to map integers to reals.

7. Markov Chain

The first major component of the DFT specification that we present is the Markov chain. In this section we specify those aspects of a Markov chain type that we need to express the rest of our specification.

\[
\begin{align*}
\text{MarkovState} \\
\text{MarkovTrans} & \triangleq \text{MarkovState} \times \text{MarkovState}
\end{align*}
\]

We introduce a given set \( \text{MarkovState} \) and a type alias \( \text{MarkovTrans} \), which is simply a relation between Markov states. These types define the states and transitions of a Markov chain.

\[
\begin{align*}
\text{MarkovChain} \\
\text{states} : \mathbb{F} \text{MarkovState} \\
\text{transitions}s : \mathbb{F} \text{MarkovTrans} \\
\text{initStPrObs} : \text{MarkovState} \rightarrow \mathbb{R} \\
\text{finalStPrObs} : \text{MarkovState} \rightarrow \mathbb{R} \\
\text{rates} : \text{MarkovTrans} \rightarrow \mathbb{R}
\end{align*}
\]

\[
\begin{align*}
\text{dom} \text{transitions} & = \text{states} \\
\text{ran} \text{transitions} & = \text{states} \\
\text{dom} \text{initStPrObs} & = \text{states} \\
\text{dom} \text{finalStPrObs} & = \text{states} \\
\forall p : \text{ran} \text{initStPrObs} \cdot 0_\mathbb{R} \leq p \leq 1_\mathbb{R} \\
\forall p : \text{ran} \text{finalStPrObs} \cdot 0_\mathbb{R} \leq p \leq 1_\mathbb{R} \\
\text{dom} \text{rates} & = \text{transitions} \\
\forall r : \text{ran} \text{rates} \cdot r \geq 0_\mathbb{R}
\end{align*}
\]

A Markov chain (in our formulation) comprises a set of states, a set of transitions between states, a set of initial state probabilities, a set of final state probabilities, and a set of transition rates. The first two predicates (7.1) state that the transitions must be over the particular set of states \( \text{states} \). The next four predicates (7.2) state that the initial and final state probability functions must have as their domain \( \text{states} \), and that the probabilities must range from 0 to 1 inclusive. The final pair of predicates (7.3) state that the \( \text{rates} \) function must have as its domain the transitions of the Markov chain, and the rates must be greater than or equal to 0.

The semantics of the Markov chain are incomplete—we have not specified how the final state probabilities are computed from the transition rates and the initial state probabilities. However, Markov chains are well enough understood that we are willing to elide the details.

8. Failure Automaton

Having specified the Markov chain, we now specify the failure automaton. Our overall specification will make use of an \( \text{Event} \) type to represent events in a fault tree—basic events or event associated with gates. We delay specification of the details of events until Section 10. However, one detail, \( \text{EventID} \), is used in this section. An \( \text{EventID} \) is a unique identifier for an event that pertains to that event even as it undergoes state changes.

\[
\text{FailureAutomaton} \triangleq \text{FaultTree}
\]

A failure automaton state is a fault tree. In the current version of our specification, a fault tree consists both of the state-independent structural elements such as the identities of inputs to gates, as well as the state-dependent attributes such as spare allocation and event histories.

\[
\begin{align*}
\text{FailureAutomaton} & \triangleq \\
( \text{FailureAutomatonState} \times \text{EventID} ) \times \text{FailureAutomatonState}
\end{align*}
\]

We define a \( \text{FailureAutomatonTrans} \) as a mapping from \( \text{FailureAutomatonState} \) and \( \text{EventID} \) to \( \text{FailureAutomatonState} \). The \( \text{EventID} \) is the identifier for the basic event whose occurrence causes all other event occurrences in the destination state that had not occurred in the originating state.

\[
\begin{align*}
\text{FailureAutomatonState} : & \mathbb{F} \text{FailureAutomatonState} \\
\text{transitions} : & \mathbb{F} \text{FailureAutomatonTrans} \\
\text{sysFailCovStates} : & \mathbb{F} \text{FailureAutomatonState} \\
\text{sysFailUncovStates} : & \mathbb{F} \text{FailureAutomatonState}
\end{align*}
\]

\[
\begin{align*}
\forall f_1, f_2 : \text{states} \cdot \\
\text{FailureAutomatonStatesConsistent}(f_1, f_2) \\
\text{sysFailCovStates} = \\
\{ s : \text{states} \mid s.\text{status} = \text{failedCovered} \} \\
\text{sysFailUncovStates} = \\
\{ s : \text{states} \mid s.\text{status} = \text{failedUncovered} \}
\end{align*}
\]

\[
\forall t : \text{transitions} \cdot \\
\text{GetFromState}(t) \in \text{states} \land \text{GetToState}(t) \in \text{states} \land \\
\text{FailureAutomatonTransitionConsistent}(t)
\]
A failure automaton comprises a set of states, some of which are system failure states, and a set of transitions. The first predicate states that any two states in the failure automaton must describe the state of the same fault tree (i.e., the state-independent parts match). The second predicate uses set comprehension to define the covered and uncovered failure states, based on whether the state is failed as the result of some sequence of basic event failures, or whether the state is failed as the result of the single point failure of a basic event.

GetFromState and GetToState (not presented here) compute the originating and destination failure automaton states, respectively, for the transition. The third predicate ensures that the transitions map from states to states, and that the transition is valid. In the interest of space, we elide the FailureAutomatonStateConsistent function. However, we do elaborate FailureAutomatonTransitionConsistent:

FailureAutomatonTransitionConsistent_: FailureAutomatonTransition Consistent:

∀ t : FailureAutomatonTransition:
  GetFromState(t).history = front(GetToState(t).history) ∧
  SparingConsistent(GetFromState(t), GetToState(t)) ∧
  IsACausalBasicEvent(GetToState(t), GetCausalEventID(t))

Transitions are valid if three conditions hold. The first condition is that the destination state has a history whose prefix is the history of the source state. The second condition ensures valid spare allocation. SparingConsistent, not presented here, states that (1) a failed spare gate not using a spare continues to not use a spare, (2) a spare gate whose spare being used did not fail continues to use that spare, and (3) that if a spare was in use, then either no spare is in use in the next state, or a later spare is used. Further constraints on the spares that can be used for a given fault tree will be given later in the specification of the spare gate.

Finally, IsACausalBasicEvent (specification not shown) ensures that the event whose EventID is associated with the transition is one that could have caused all of the newly occurred events. GetCausalEventID computes the identifier of the “causal event” that is responsible for the change in the state in the failure automata. The set of newly occurred events is the total set of events that have occurred in the next state of the fault tree state machine, which were not occurred in the previous state. This aspect is important because it says that no new events occur in the latter state unless they were caused, directly or indirectly, by the triggering event.

This definition of FailureAutomatonTransitionConsistent resolves Issue #5 by introducing nondeterminism in the next state. There are possibly multiple valid ways to satisfy the SparingCorrect condition for a given source state and causal event.

9. Semantics of Failure Automata

Having specified FailureAutomata and MarkovChain, we now specify, in two parts, the semantics of a particular subtype of failure automata in terms of Markov chains. The subtype, WeibullOrExponentialFailureAutomaton, whose precise definition is elided, comprises those failure automata whose basic event probability distributions are Weibull or exponential only. We begin by specifying the general constraints on the functions that map FA states and transitions to Markov chain states and transitions.

WeibullOrExponentialFailureAutomatonToMarkovChain1

autStToMarkSt : FailureAutomatonState → MarkovState
autTrToMarkTr : FailureAutomatonTransition → MarkovTransition

∀ st : MarkovState, dom autStToMarkSt = autStToMarkSt(st)
∀ tr : FailureAutomatonTransition, dom autTrToMarkTr = autTrToMarkTr(tr)

For a particular failure automaton fAutomaton and its corresponding Markov chain mChain, there is a bijection autStToMarkSt which maps each failure automaton state to a unique Markov state. autTrToMarkTr maps one or more failure automaton transitions to a Markov transition. As described in Issue #7, different event occurrences can yield the same resulting failure automaton state from a particular failure automaton state. As a result, there may be multiple arcs between two states in the failure automaton.

We now specify the detailed constraints on the state and transition mappings:

WeibullOrExponentialFailureAutomatonToMarkovChain2

mTime : MissionTime
systemFailureProbability : Real

∀ fa : FailureAutomaton, mTime : MissionTime
  systemFailureProbability

∀ s : mChain.states, fs : FailureAutomatonState
  mChainRates(fs, s) ≤ IntToReal(systemFailureProbability)

∀ s : mChain.states
  mChainRates(s, mChain.initStProbs) ≤ IntToReal(systemFailureProbability)

∀ fs : FailureAutomatonState, s : mChain.states
  mChainRates(fs, s) ≤ IntToReal(systemFailureProbability)

∀ fa : FailureAutomaton, tr : FailureAutomatonTransition
  autStToMarkSt(faGetFromState(tr)) ≤ IntToReal(systemFailureProbability)

∀ fA : FailureAutomaton, s : mChain.states
  autStToMarkSt(fAGetToState(tr)) ≤ IntToReal(systemFailureProbability)

∀ fs : FailureAutomatonState, tr : FailureAutomatonTransition
  autTrToMarkTr(fs, tr) ≤ IntToReal(systemFailureProbability)

∀ fa : FailureAutomaton, s : mChain.states
  autTrToMarkTr(faGetFromState(tr), faGetToState(tr)) ≤ IntToReal(systemFailureProbability)
GetEvent (elided) computes the event in a fault tree that corresponds to an event identifier.

Predicate 9.1 states that each transition in the failure automaton that maps from \(fs\) to \(ts\) must correspond to the Markov transition that maps states in the Markov chain which correspond to \(fs\) and \(ts\).

Predicate 9.2 assigns initial state probabilities for the Markov chain. \(is\) is the set of initial states—those that aren’t the destination state of a transition. If \(st\) is an initial state in the Markov chain, then it has an initial state probability equal to one divided by the total number of starting states. If \(st\) is not an initial state, then it has an initial state probability equal to zero.

Predicate 9.3 specifies that the rates for the Markov chain are computed from the failure automaton using the ComputeTransitionRate function described below. Predicate 9.4 states that the system failure probability is equal to the sum of the final state probabilities for all of the system failure states.

ComputeTransitionRate, not described here, computes the transition rate for each Markov transition based on the characteristics of the failure automaton transitions and the mission time. There are four components of the Markov transition rate. The first is the hazard rate of the basic event, which is computed based on the basic event’s failure distribution. The second is the sparing scale factor, which accounts for multiple non-deterministic states resulting from spare gate contention for spares (Issue #2). The sparing scale factor evenly distributes the rate of transition by dividing the transition rate by the number of (non-deterministic) next states. The third aspect of the transition rate computation is the coverage factor, which accounts for uncovered failures, where basic events can have a small probability of causing system failure irrespective of the failure relationships modeled by the fault tree. The final component is a replication scale factor that both adjusts for the increased rate resulting from additional components, and adjusts for basic event replicates which may be unused spares failing at a reduced rate.

10. Fault Tree

In this section we provide the specification of AND, PAND, and spare gates, as well as the specification of basic events. We elide the specification of the OR, KOFM, and FDEP constructs. Lastly, we specify the fault tree itself.

10.1. Events and Basic Events

An event can either be a gate or a basic event. The replication for gates will be constrained to 1. Note that in the face of replication, a basic event can be “not occurred”, where \(numberOccured = 0\), “partially occurred”, where \(0 < numberOccured < replication\), or “fully occurred”, where \(numberOccured = replication\).

\[
\begin{array}{l}
\text{Event} \\
\quad \text{id} : \text{EventID} \\
\quad \text{replication} : \mathbb{N}_1 \\
\quad \text{numberOccured} : \mathbb{N} \\
\end{array}
\]

\[
\text{numberOccured} \leq \text{replication}
\]

This schema formalizes the notion of an event, which has an identifier, a replication value (Issue #1), and a state-dependent component for the number of occurred replicates.

\[
\begin{array}{l}
\text{BasicEvents} \\
\quad \text{basicEvents} : \mathbb{F} \text{ Event} \\
\quad \text{dormancies} : \text{Event} \rightarrow \mathbb{R} \\
\quad \text{coverages} : \text{Event} \rightarrow \text{CoverageModel} \\
\quad \text{distributions} : \text{Event} \rightarrow \text{Distribution} \\
\end{array}
\]

\[
\begin{array}{l}
\quad \forall \text{ be : basicEvents} \\
\quad \text{dormancies(be)} \geq 0
\end{array}
\]

The BasicEvents schema modularizes the state and predicates of a fault tree that are related to basic events. In particular, there is a set of events that are basic events, as well as a dormancy, coverage, and distribution for each basic event. The predicates state that dormancy, coverage models and distributions are defined only for basic events, and that all dormancy values are non-negative.

10.2. AND Gates

The gates in a fault tree are simply a finite set of events whose state is specified in terms of an associated sequence of inputs. For the combinatorial AND, OR, and KOFM gates we overspecify by using a sequence of inputs instead of a set because this allows us to treat inputs to gates uniformly. Input ordering is strictly needed only for dynamic gates, like PAND, that are order-sensitive.

Below is the specification of the AND gate:

\[
\begin{array}{l}
\text{AndGates} \\
\quad \text{andGates} : \mathbb{F} \text{ Event} \\
\quad \text{inputs} : \text{Event} \rightarrow \text{seq(Event)} \\
\end{array}
\]

\[
\begin{array}{l}
\forall \ g : \text{andGates} \\
\quad \text{g.replication} = 1 \wedge \\
\quad \text{g \in \text{dom inputs}} \wedge \\
\quad \forall \ e : \text{ran(inputs}[g]) \ |
\quad \text{e.numberOccured = e.replication} \Rightarrow \\
\quad \text{g.numberOccured = 1} \wedge \\
\quad \neg \forall \ e : \text{ran(inputs}[g]) \\
\quad \text{e.numberOccured = e.replication} \Rightarrow \\
\quad \text{g.numberOccured = 0}
\end{array}
\]

\[
\text{numberOccured} \leq \text{replication}
\]
**andGates** is the set of all AND gates in a fault tree. **inputs** is a partial function from events to non-empty sequences of events that do not contain repeated elements. **inputs** is a partial function because basic events are events but do not have inputs. Recall that because we are using schemas for modularization, the **inputs** function in this schema and later gate schemas are the same when we later conjoin them in the FaultTree specification using schema inclusion. That is, the domain of **inputs** will be the union of all the gates in the fault tree.

Predicate 10.2 states that each AND gate must have a replication of 1. This is a requirement that we will impose for all gates, in keeping with the current widely accepted definitions. However, this specification accommodates the possibility of later expanding the notion of replication to include gates.

Predicate 10.3 states that every AND gate must have an associated sequence of inputs. The last two predicates define the state of the AND gate with respect to the inputs. Predicate 10.4 states that an AND gate occurs if all of the inputs are fully occurred, and predicate 10.5 states that it does not occur otherwise.

### 10.3. PAND Gates

In this section we first formalize the notion of a history of events, and then use that specification to provide a semantics for PAND gates.

**CurrentStateOfEvents** ≡ \( \forall \text{Event} \)

**CurrentStateOfEvents** represents the state of all the events for a fault tree. Note that this does not capture the entire state of the fault tree; in particular, the state of spare gates using spares is not modeled.

\[
\text{History} ≡ \{ h : \text{seq CurrentStateOfEvents} | \forall i,j : [1..\#h] . \text{GetIDs}(h[i]) = \text{GetIDs}(h[j]) \}\]

A **History** (Issue #4) is specified here as a sequence of **CurrentStateOfEvents** (without repeated elements) that represents the changing state of the fault tree over a set of discrete time steps. Every step in the history has the same set of events, although the event states can change (the set of event identifiers is constant, to be precise).

Note that many histories are invalid in the context of a particular fault tree, because the fault tree semantics impose certain constraints. For example, an event associated with an AND gate must occur when all input events have occurred. Valid histories are also subject to the constraints imposed by spare allocation and functional dependencies.

We can now specify the semantics of the PAND gates in a fault tree. Recall that the inputs to a gate may be replicated basic events. We must make a distinction between the in-order failure of the inputs, versus the in-order failure of the replicates. As mentioned earlier, Issue #1 regarding the lack of ordering of replicates complicates the semantics of the PAND.

There are several possible semantics that can be used in this case. For example, the inputs could be considered in-order if the first failure of each replicated input occurs in order. The semantics we have chosen is that the inputs fail in order if all the replicates in position \( i \) fail in order with respect to all the replicates in positions after \( i \).

Another complication in the semantics of the PAND is whether the term “in order” means strictly in order or not. In this specification and in the following discussion ordering is not strict. That is, the simultaneous failure of any two replicates is considered to be “in order”, even if the two replicates are in different inputs to the gate.

**PandGates**

\[
pandGates : \exists \text{Event} \\
\text{inputs} : \text{Event} \rightarrow \text{iseq Event} \\
\text{history} : \text{History} \\
\]

\[
\forall g : \text{pandGates} . \\
g . \text{replication} = 1 \land \\
g \in \text{dom inputs} \land \\
\text{InputsOccurredAndInOrder}(\text{history}, \text{inputs}(g)) \Rightarrow \\
g . \text{numberOccurred} = 1 \land \\
\neg \text{InputsOccurredAndInOrder}(\text{history}, \text{inputs}(g)) \Rightarrow \\
g . \text{numberOccurred} = 0 \\
\]

In the specification above, the gate occurs if the inputs are all failed, and they all failed in order. The gate does not occur otherwise. Note the additional state component **history**, which is used by **InputsOccurredAndInOrder**.

\[
\forall h : \text{History}; s : \text{iseq Event} . \\
\text{InputsOccurredAndInOrder}(h, s) \Leftrightarrow \\
(\forall i : [1..\#s] . s[i].\text{numberOccurred} = s[i].\text{replication} \land \\
(\forall i : [1..\#s] . s[i].\text{numberOccurred} = s[i].\text{replication} \land \\
\text{FirstFailureTime}(h, s[i].id) \leq \text{FirstFailureTime}(h, s[i+1].id)) \\
\]

Given a history and a sequence of events, the value of the **InputsOccurredAndInOrder** function is true if the replicates in each position fail before or at the same time as the replicates in later positions. Predicate 10.6 states that all the inputs must be fully occurred, and predicate 10.7 states that the event at position \( i \) must be fully occurred at or before the time at which the first replicate at position \( i + 1 \) occurs. This specification resolves Issue #1 involving the lack of ordering of basic event replicates, and Issue #6 involving the case of simultaneous failure of replicates of different inputs.

### 10.4. SpareGates

In this section we present the gate-level specification of the spare gate, which specifies the state of the gate and in-
variants on the state regarding the use of spares, independent of the state of the overall fault tree.

\[
\text{Spare Gates}
\]

\[
\text{spareGates} : \forall \text{Event} \\
\text{inputs} : \text{Event} \rightarrow \text{Seq}_1 \text{Event} \\
\text{spareInUse} : \text{Event} \rightarrow \text{Event}
\]

\[
\forall \ g \ : \ \text{spareGates} \cdot g \cdot \text{replication} = 1 \quad \quad (10.8) \\
\text{dom} \text{spareInUse} \subseteq \text{spareGates} \quad \quad (10.9) \\
\forall \ g \ : \ \text{spareInUse} \in \text{spareGates} \forall \ g \in \text{ran} \{ \text{inputs}(g) \} \quad \quad (10.10) \\
\forall \ s : \ \text{spareInUse} \in \text{spareGates} \forall \ NumSpGatesUsingSpare(\text{spareInUse}, s.\text{id}) \leq e.\text{replication} - e.\text{numbOccCurrent} \quad \quad (10.11) \\
\forall \ g_1 : \ \text{dom} \text{spareInUse}, i : \forall_1 \{/ \ \\
\text{spareInUse}(g_1) = \{ \text{inputs}(g_1) \}[i] \} \quad \quad (10.12) \\
\forall \ j : 1 \ldots i - 1; e : \text{Event} \in \text{dom} \text{spareInUse}[g_1] \in \text{inputs}(g_1)[j] \} \ \\
\text{NumSpGatesUsingSpare}(\text{spareInUse}, e.\text{id}) = e.\text{replication} - e.\text{numbOccCurrent} \quad \quad (10.13) \\
\forall \ g_1 : \ \text{spareGates} \ /
\forall \ e : \ \text{ran} \{ \text{inputs}(g_1) \}[i] \} \\
\text{NumSpGatesUsingSpare}(\text{spareInUse}, e.\text{id}) = e.\text{replication} - e.\text{numbOccCurrent} \quad \quad (10.14) \\
\forall \ g : \ \text{spareGates} \ \\
\{ g \in \text{dom} \text{spareInUse} \Rightarrow g.\text{numbOccCurrent} = 0 \wedge \ \\
g \notin \text{dom} \text{spareInUse} \Rightarrow g.\text{numbOccCurrent} = 1 \}
\]

Spare gates are events having associated inputs. \text{spareInUse} is a partial function that specifies which spare is in use by a spare gate. The function is partial because not all events are spare gates, and spare gates use no spare when they are failed. \text{NumSpGatesUsingSpare} (elided) specifies the number of spare gates using replicates of a given replicated basic event. The predicates are explained as:

- (10.8) Every spare gate has a replication of 1.
- (10.9) Only spare gates can use spares from a pool. The domain of \text{spareInUse} is not equal to \text{spareGates} because failed spare gates do not use any spare.
- (10.10) Every spare gate that uses a spare uses one from a pool that is in its input sequence.
- (10.11) The number of spare gates using any pool of spares must be less than or equal to the number of operational replicates of the pool.
- (10.12) For every spare gate that is using a spare pool in a particular position in its input list, all pools before that pool in the inputs list must be fully utilized. (All replicates are in use by other spare gates or are failed.)
- (10.13) For all spare gates that are not using any spare from any pools, each of the pools must be fully utilized. (All replicates are in use by other spare gates or are failed.)
- (10.14) The gate is operational if a spare is being used, and failed otherwise.

There are two major differences between the spare gate specification presented here and the previous, informal definitions of the CSP, WSP, and HSP gates. The first is that the previous definitions of spare gates stipulated that a particular input, called the primary, would be used first. Constraints were placed on the primary in order to ensure that the CSP, WSP, or HSP would always be operational in the starting state of the failure automaton. We removed this restriction in order to provide a more abstract spare gate whose inputs are treated uniformly.

The second change is that there is no longer a "temperature" (cold, warm or hot) for the spare gate. Instead, the attenuation of the failure rate of an unused, unfailed replicate of a basic event is dictated solely by the dormancy. This change resolves Issue #3, providing more orthogonality between spare gates and basic events and removing restrictions on sharing of spares among spare gates.

### 10.5. The Fault Tree

We now present the Fault Tree specification, which places constraints on the valid integration of fault tree elements. This specification completes the specification of the failure automaton described earlier. We begin by presenting the constraints on the gate events and basic events:

\[
\text{Fault Tree} : \quad \text{AND Gates} \\
\text{OR Gates} \\
\text{Kofm Gates} \\
\text{Pand Gates} \\
\text{SEQs} \\
\text{FDEPs} \\
\text{Basic Events} \\
\text{spareGates} : \forall \text{Event} \\
\text{events} : \forall \text{Event} \\
\text{gates} = \{ \text{basicEvents}, \text{andGates}, \text{orGates}, \text{kofmGates}, \text{pandGates}, \} \quad \quad (10.15) \\
\text{spareGates} = \text{events} \setminus \text{basicEvents} \quad \quad (10.16) \\
\forall e_1, e_2 : \text{events} \Rightarrow e_1.\text{id} = e_2.\text{id} \Rightarrow e_1 = e_2 \quad \quad (10.17) \\
\text{systemEvent} \in \text{events} \quad \quad (10.18) \\
\text{systemEvent} \notin \text{events} \setminus \{ i : \text{ran} \text{inputs} \cdot \text{ran} i \} \quad \quad (10.19) \\
\text{systemEvent}.\text{replication} = 1
\]

The Fault Tree scheme uses schema inclusion to combine the schemas for the various kinds of gates and basic events into an overall specification. Identically named state components in the included schemas are merged in the resulting schema, and all of the predicate parts of the included schemas are implicitly included and conjoined. The additional constraints stipulated here state the following:

- (10.15) An event is either a basic event, an AND gate,
an OR gate, a KOFM gate, a PAND gate, or a spare gate.

- (10.16) Gates are all the events in a fault tree except for the basic events.
- (10.17) Event identifiers are unique
- (10.18) The system failure event must be from the set of events, and it cannot be an input to any gate.
- (10.19) The last system state in the history must match the current state.

We now present the remainder of the fault tree specification, using schema inclusion to extend the previous specification.

\[
\text{Fault Tree}
\]

\[
\text{Fault Tree 1}
\]

\[
\text{systemEvent} : \text{Event}
\]

\[
\text{status} : \text{Fault Tree Status}
\]

\[
\forall g : \text{events} \mid g \neq \text{systemEvent} \quad \text{IsInputTo}(\text{inputs}(g), \text{systemEvent}) \tag{10.20}
\]

\[
\forall g : \text{spare Gates} \mid \text{ran}(\text{inputs}(g)) \subseteq \text{basic Events} \tag{10.21}
\]

\[
\forall g_1 : \text{spare Gates}, g_2 : \text{gates} \setminus \text{spare Gates}; \quad i_1 : \text{basic Events} \quad i_2 : \text{basic Events} \quad i_1 \in \text{ran}(\text{inputs}(g_1)) \quad i_2 \notin \text{ran}(\text{inputs}(g_2)) \tag{10.22}
\]

\[
\forall s : \text{segs} \mid \text{ran}(s) \subseteq \text{events} \tag{10.23}
\]

\[
\text{dom fdeps} \subseteq \text{events} \tag{10.24}
\]

\[
\forall d : \text{run fdeps} \mid d \subseteq \text{basic Events} \tag{10.25}
\]

\[
\text{systemEvent. numberOccurred} = 0 \Rightarrow \text{status} \neq \text{operational} \tag{10.26}
\]

Finally, The Fault Tree schema extends the Fault Tree 1 schema given above. A fault tree can be in one of three major states: operational, failed as a result of the occurrence of the system-level failure event, and failed due to the occurrence of an uncovered failure that “takes down” the whole system. The state component systemEvent identifies the system-level failure event, while status expresses which of the three possible major states a fault tree is in.

The predicates are explained as:

- (10.20) All gates must be connected to the top level gate. IsInputTo (elided) specifies whether one event is an input to another, whether directly or indirectly.
- (10.21) The inputs to each spare gate must be basic events. This is an area where the specification could possibly be generalized to allow any type of gate as a spare.
- (10.22) Inputs to spare gates can not be inputs to other gates. Note that inputs to spare gates are allowed to be inputs to other spare gates, and inputs to functional dependencies.
- (10.23) The inputs to each sequence enforcer must be events in the fault tree.
- (10.24) The triggers for functional dependencies must be events in the fault tree.
- (10.25) The dependent inputs to the functional dependencies must be basic events.
- (10.26) A fault tree can not be operational if the system level event has occurred.

11. Semantics of Fault Trees

In this section, we complete the definition of the mapping from DFTs to FAs. Every valid fault tree has a corresponding failure automata. In previous sections we have specified the notions of Fault Tree and Failure Automaton. Given these semantic domains, the mapping between them is straightforward:

\[
\text{Fault Tree To Failure Automaton}
\]

\[
\text{faultTree} : \text{Fault Tree}
\]

\[
\text{failureAutomaton} : \text{Failure Automaton}
\]

\[
\forall fas : \text{failure Automaton. states} \quad \text{Failure Automaton States Consistent}[\text{fault Tree, fas}]
\]

Because each failure automata state is simply a fault tree in a particular failure configuration, we need only establish that the state-independent components of the failure automaton states correspond to the state-independent components of the particular fault tree whose semantics we wish to express. We use the Failure Automaton States Consistent relation to ensure that the fault tree matches each of the states, disregarding event failures, spare allocation, and other state-dependent aspects.

12. Related Work

The semantics of static gates and basic events had previously been defined informally in terms of probability theory [10]; and dynamic gates have been specified informally in terms of Markov chains [3, 5, 6, 8]. However, to the best of our knowledge, our work is the first attempt to present a mathematically precise, abstract, and reasonably complete semantics for dynamic fault trees (or for static fault trees, for that matter).

In earlier work we used Z to define precisely the meaning of certain dynamic fault tree gates, and the structure of a dynamic fault tree [4]. However, we did not attempt to formalize the semantics of a dynamic fault tree as a whole in terms of an underlying semantic domain (e.g., Markov chains) as we do in this paper.
Our work is analogous to that of Abowd, Allen, and Garlan [1], who used Z to formalize software architectural diagramming notations. They use Z to formally define the components and connectors typically used in such diagrams. Using this formalism, the authors then use the framework to perform analyses both within and across architectural styles. We observe that there is a need for formal semantics for modeling frameworks used in a wide variety of engineering tools; and we provide a semantics for one such framework in particular.

13. Conclusion

Mitigating the risk that adverse engineering decisions will be made on the basis of invalid results produced by computational modeling and analysis tools requires that the semantics of the modeling frameworks that such tools implement be defined precisely, completely in areas of uncertainty, and abstractly. The existence of such specifications is necessary for the validation of such modeling frameworks, as a basis for sound user documentation, as a target for program implementation and verification, and as a definitive definition of the scientific basis of the modeling method.

None of the kinds of specifications that are most commonly used today—natural language, semantics for selected special cases, and source code—can adequately meet these requirements. Formal methods for precise specification exist; yet, formal semantic definitions for modeling frameworks for engineering are rarely presented. Nor do software engineering and languages researchers fully understand how to create conditions under which these techniques can be used profitably.

We have presented an approach to addressing this problem based on multi-disciplinary collaboration between specification and domain experts. We used the specification language Z to define semantics in the denotational style for a framework for dynamic fault tree analysis that supports several subtle modeling constructs. An important future task is a rigorous review of the specification with a group of experts in reliability engineering to more fully validate the specification. We also plan to undertake some formal validation using theorem proving tools to aid in the formal proof that the specification has certain desired properties.

The results of this work fall into two categories. First, we present the first fully formal semantics for an important new modeling framework, thus strengthening the scientific foundations for that work, and providing a basis for the construction of trustworthy software tools. Second, we have shown that it is feasible to employ existing formal specification techniques to develop precise, abstract specifications for practical frameworks. Our specification is not fully complete. For example, we have no axiomatization of real or computer arithmetic. Nor have we fully specified important aspects of a modular approach to dynamic fault tree analysis in which complex trees are broken into parts and analyzed separately. However, our work appears to demonstrate both the importance and the practicality of developing such a specification. We are now developing this more complete specification.

Acknowledgements

This work was supported in part by the National Science Foundation under grant CCR-9804078 (Kevin Sullivan, principal investigator).

References