

Shared Semantic Domains for Computational Reliability Engineering

David Coppit

Dept. of Computer Science
The College of William and Mary
Williamsburg, VA 23188
david@coppit.org

Robert R. Painter

Dept. of Computer Science
The College of William and Mary
Williamsburg, VA 23188
rrpain@cs.wm.edu

Kevin J. Sullivan

Dept. of Computer Science
The University of Virginia
Charlottesville, VA 22903
sullivan@virginia.edu

Abstract

Modeling languages and the software tools which support them are essential to engineering. However, as these languages become more sophisticated, it becomes difficult to assure both the validity of their semantic specifications and the dependability of their program implementations. To ameliorate this problem we propose to develop shared semantic domains and corresponding implementations for families of related modeling languages. The idea is to amortize investments at the intermediate level across multiple language definitions and implementations. To assess the practicality of this approach for modeling languages, we applied it to two languages for reliability modeling and analysis. In earlier work, we developed the intermediate semantic domain of failure automata (FA), which we used to formalize the semantics of dynamic fault trees (DFTs). In this paper, we show that a variant of the original FA can serve as a common semantic domain for both DFTs and reliability block diagrams (RBDs). Our experiences suggest that the use of a common semantic domain and a shared analyzer for expressions at this level can ease the task of formalizing and implementing modeling languages, reducing development costs and improving their dependability.

1. Introduction

Computational modeling and analysis is now essential in all engineering disciplines. Engineers use domain-specific languages to create system models, which are then analyzed to infer certain system properties. In order to model more complex systems, researchers are developing increasingly sophisticated languages whose constructs correspond more closely to concepts in the domain.

For example, reliability engineers developed fault tree models [18, 19, 20] in the 1960's to model combinations of failures in complex systems. In the 1990's increasingly complex, fault-tolerant, computer-based systems led

researchers to extend the fault tree language to support the modeling of order-dependent failures [3, 9]. More recent work has added the ability to model uncovered failures [11], and to model systems having multiple phases of operation [21]. Such advances are meant to provide engineers with a high-level language for creating models that more accurately represent the system being modeled.

As the feature sets and levels of expressiveness of such languages increase, however, it becomes difficult for language designers to be sure that language semantics are well-defined and validated. Interactions among language features and the widening *semantic gap* between a language and its underlying mathematical semantics make it difficult to reason about the meaning of models. Software developers must also address these issues when developing tools to support the modeling language, and, in particular, when writing routines to translate models in the high-level language into mathematical representations (e.g., differential equations) to be analyzed by standard subroutine libraries.

Today, even when several modeling languages are fairly closely related, precise semantic specifications are rare, and it is unclear that much attempt is made to leverage verification efforts by using verified, shared source code bases. There are several consequences of this problem. First, research is stymied by the amount of effort required to develop, validate, and deploy new modeling languages. Second, new advances must be integrated separately into each language and implementation. Lastly, and most importantly, engineers are exposed to an increased risk of error in the definition and implementation of the languages. This is particularly troublesome because analysis results from high-level models can be difficult for engineers to validate.

In this paper we present an approach to the specification, validation and implementation of modeling languages that shows some promise of ameliorating such problems. The approach is to develop a common intermediate language, whose specification both defines a shared semantic domain for multiple high-level languages, and provides the basis for shared analysis programs. The idea is to define the mean-

ing of models in multiple high-level languages in terms of expressions at this common intermediate level, and to use a single shared solver implementation for those expressions.

This intermediate language, as a conceptual construct, eases the definition of high-level modeling languages by reducing the semantic gap between these languages and their lower-level representations. It also provides the specification for a stable, debugged, reusable library for implementing high-level languages. Effort expended in specifying and validating the intermediate language is amortized across multiple high-level languages. For example, one can invest effort in *formally* specifying the intermediate language, which is then used to help define the high-level languages.

In earlier work, we developed the intermediate language of *failure automata* (FAs) as part of an effort to formalize, validate, and implement the semantics of dynamic fault trees (DFTs) [4, 6, 7]. We formalized the semantics of DFTs in terms of FAs, and mapped a subset of FAs to Markov chains. In this way, we formalized the semantics of the subset of DFTs having Markovian interpretations (not all DFTs do), while creating options to extend our formalization to DFTs whose FAs do not have Markovian semantics.

The current work is based on the observation that FAs might serve as a common semantic domain for reliability modeling languages other than DFTs. To test this idea, we explored the use of FAs as an intermediate semantic domain for *reliability block diagrams* (RBDs) in addition to DFTs. The contribution of this work is not the use of intermediate languages—this has been done for years in the programming language community [10]. Rather, we demonstrate that the FA, with some adjustments, does provide a useful intermediate domain for specifying and implementing both DFTs and RBDs. This data point provides some support for the broader hypothesis that common intermediate languages might be developed for modeling languages more generally.

The rest of this paper is organized as follows. Section 2 provides background information on reliability modeling languages and the formal specification language we have used in our research. Section 3 presents the formal definition of the failure automaton. Sections 4 and 5 present our formal definition of RBDs and DFTs in terms of FAs. Section 6 presents an summary of the DFT and RBD implementations. In Section 7 we evaluate our work. Section 8 discusses related work, and Section 9 concludes.

2. Background

Reliability engineering researchers have developed a number of languages for modeling system features such as combinations and orders of failures, redundant components, failure dependencies, component repair, and uncovered failures. Examples include reliability block diagrams, fault trees, reliability graphs, Markov chains, Petri nets, Markov

reward models, and stochastic reward nets. There are also variants of these languages; RBDs, for example, may be directed or undirected, and may or may not support multiple blocks for the same component.

At a high level, each of these notations represents relationships between component failures (or more generally, any event) in a system. Most notations support reliability modeling concepts such as redundancy, sparing behavior, and common-cause failures. The occurrence of events such as a basic component failure is modeled using probability distributions whose characteristic parameters are often derived from field data. Finally, each language has criteria for evaluating the status of the overall system model with respect to individual event occurrences.

Generally speaking, such languages are either combinatorial-based or state-based. From the engineer’s perspective, combinatorial-based languages can model combinations of event occurrences, while state-based languages can also model permutations. In this sense combinatorial-based languages are weaker than state-based languages. However, models expressed in state-based languages are restricted to Markovian (i.e. memoryless) distributions of component failures. Models expressed in state-based languages are also more computationally difficult to analyze, which can limit their practical size.

Software tools implement the semantics of reliability languages in terms of analysis routines. Such routines can automatically compute properties of the model such as the overall system unreliability, or the sensitivity of the unreliability relative to particular basic events. The difficulties of correctly implementing the software are compounded by the semantic complexities of the reliability languages.

2.1. Reliability Block Diagrams

Reliability block diagrams (RBDs) [1] are a widely used notation in engineering. An RBD consists of multiple blocks with input and output ports connected by lines. A set of blocks represent a component in the system being modeled, and a block completes the path from its input port to its output port if its corresponding component is operational. There are two unique blocks, the *source* and *sink* which are not associated with any component. The system is considered to be operational if there exists a path through the network of blocks from the source to the sink.

RBDs are a combinatorial notation. Two blocks connected in series means that the path will be broken if the component associated with either block fails. Two blocks connected in parallel means that there are two viable paths, and the components for both blocks must fail for the pair of paths to be broken. RBDs can also be hierarchical, such that any given block may be expressed in terms of a more detailed RBD.

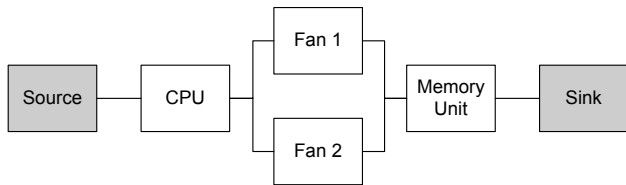


Figure 1. Example reliability block diagram

Figure 1 shows an example RBD. In this RBD, the CPU and memory units are connected in series with each other and a pair of redundant fans, which are connected in parallel. The fans are redundant in that if only one of them fails, there will still be a path from the source to the sink. Overall, the system will be operational as long as the CPU and memory unit are operational, as well as at least one of the fans.

Each block is modeled using probability distributions. Software tools analyze the reliability of the system for a given time t using the reliability for each block at time t . The overall system reliability of the RBD is computed by translating it into a probabilistic Boolean expression. This expression, along with the block reliabilities, can be solved using standard methods.

There are multiple variants of RBDs. Directed RBDs have directed lines which limit the effect of block failures. A special case of RBDs only allows one block to represent a component. Some variants also allow for multiple failure modes for blocks, such as “failed open” and “failed closed”. In this paper we will address non-hierarchical, directed RBDs with traditional operational/failed block failure modes.

2.2. Dynamic Fault Trees

Fault trees [19] are a widely used modeling notation, originally developed in the Minuteman missile program to model its control system [20]. Today we refer to such fault trees as *static* fault trees because they are combinatorial-based. More recently, researchers have extended static fault trees to support state-based semantics. One example is the *dynamic fault tree* (DFT) language, which contains constructs to model order-dependent failures [3, 9].

A dynamic fault tree is comprised of a set of *events* and *constraints*. Events can either be *basic events*, which represent component failures or the occurrence of events in the system, or *gates*, which compute a derived failure value based on some relationship in the failure of the input events. For example, an *AND* gate’s output event is considered to be failed if the input events are all failed. Basic events can also be *replicated* to indicate that a single basic event actually represents multiple identical components in the sys-

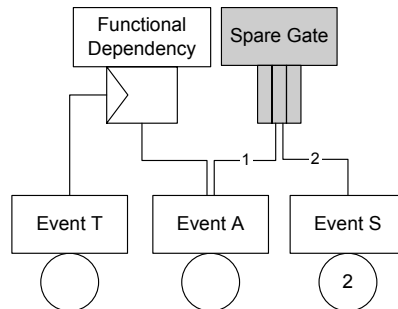


Figure 2. Example dynamic fault tree

tem. Constraints express relationships between events but do not compute failure values. For example, the *functional dependency* states that if one component fails, then all the dependent components fail as well.

Figure 2 depicts a dynamic fault tree. *Event T*, *Event A*, and *Event S* are basic events. *Event S* is a replicated basic event which represents two components. The functional dependency states that the failure of the trigger event *Event T* will cause the dependent event *Event A* to fail as well. The spare gate uses *Event A* until it is no longer available, in which case it uses a spare from *Event S*. When both replicates of *Event S* also fail the spare gate fails because there are no more redundant input basic events to utilize. The spare gate is shaded to indicate that its failure represents the overall system failure.

To build a model, the engineer connects events and constraints to express the desired failure relationships. The semantics of static fault trees were originally expressed in terms of probability theory [19], and more recently in terms of binary decision diagrams [8]. The original semantics of dynamic fault trees were illustrated in terms of translations of example DFTs to continuous-time Markov chains [3]. The semantics of Markov chains, in turn, are defined in terms of differential equations.

In previous work we presented a reasonably complete, mathematically precise definition of the semantics of dynamic fault trees [7]. At that time, the failure automaton (FA) was defined as a DFT-specific semantic domain which aided in the definition of the language. In this paper we present a revised, more general version of the FA which is more suitable for the definition of multiple reliability modeling languages.

2.3. The Z Formal Specification Notation

In this section we present a brief overview of the Z (pronounced zed) formal specification language [14]. We use this language to define the failure automaton and RBD and DFT languages in a mathematically precise way. Z supports structuring and composition of specifications expressed 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 unique set of elements. Z has a number of primitive types, such as natural numbers (\mathbb{N}) and integers (\mathbb{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 methods for user-defined types. For example, if S is a type, then $\mathbb{P}S$ is a type that comprises the set of all sets of items of type S , and $\mathbb{F}S$ is the set of all *finite sets* of type S . Instances of a type can be declared. For example, a statement such as $mySet : \mathbb{F}\mathbb{Z}$ defines a state element named “mySet” whose value is in the set of finite sets of integers.

A sequence of type S is represented as $seqS$, and an injective sequence (a sequence without repeated elements) is represented as $iseqS$. A sequence is a function, and an expression such as $mySeq(2)$ is the value of the second item in $mySeq$. $\#mySeq$ denotes the length of the sequence.

Z has many types of arrows for defining binary relations. For example, $A \rightarrow B$ is a type comprising the set of functions from the set A to the set B . Given $f : A \rightarrow B$, i.e., f is some function whose domain is a subset of A and whose range is a subset of B , $domf$ denotes the domain of f , and $ranf$ its range. Similarly, $pf : A \rightarrow B$ declares pf to be a partial function from A to B . Cross-product types are denoted by the cross operator, \times , applied to the constituent types.

Z provides a mechanism, called the schema, for the modular structuring of specifications of complex types. A schema defines a type in terms of its state components, as well as invariant relations over these state components. *Example1* below is a schema that defines a new type whose elements have two subcomponents is and j .

Example1

$is : \mathbb{F}_1 \mathbb{Z}$
 $j : \mathbb{N}$

$-3 \in is$

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 is and a natural number j . Invariant relations are stipulated in the predicate parts of schema, below the dividing line. Elements of the *Example1* type are such that the integer -3 is in is . An expression such as $ex : Example1$ states that ex is a value of type *Example1*, whose name is ex .

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

$\{e : \mathbb{Z} \mid e \bmod 2 = 0 \bullet e^2\}$

e is declared to be an integer. The $|$ symbol introduces

constraints on the locally declared variables. In this case, the remainder after dividing e by two must be zero. The statement after “ \bullet ” defines the element for the constructed set. In this case, it is the square of e .

The set of squares of even numbers, as a type, can be named in the following way:

$SquaresOfEvens \hat{=} \{e : \mathbb{Z} \mid e \bmod 2 = 0 \bullet e^2\}$

Z also supports the definition of axioms, which pertain globally to a specification. They are declared in Z in the following way:

$factorial : \mathbb{N} \rightarrow \mathbb{N}_1$ $factorial(0) = 1$ $\forall i : \mathbb{N}_1 \bullet$ $factorial(i) = i * factorial(i - 1)$
--

Here factorial is defined as a recursive function from natural numbers to non-zero 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 (\mathbb{N} subscripted with 1) in the normal way.

3. The Failure Automaton

In this section we present a formal definition of the failure automaton which has been somewhat revised and extended to better serve as an intermediate language. In sections 4 and 5 we use this revised FA to formally define the semantics of RBDs and DFTs, respectively.

The language models the system as a state machine. Each state encodes the state of events in the system, a history of past event states, the set of events which are spares in the system, the current allocation of available spares, and the overall system status. Transitions correspond to the occurrence of basic events in the system. The language supports features such as multiple initial states, nondeterministic next states, order-dependent failures, uncovered failures, and replicated basic events. The semantics of the language are formalized in terms of Markov chains. That is, each well formed failure automaton is associated with an underlying Markov chain representing its semantics.

Space constraints preclude a detailed discussion of this mapping in this paper. The reader is referred to previous publications [4, 7] which present an earlier version of the language and the complete formal semantics. Our updated version of the FA specification has been restructured to remove redundancy and corrects a few errors (one of which was discovered as a result of formalizing RBDs). We also augmented the FA to store information about the overall system status—while including this information in the FA was not necessary for DFTs, the specification of RBDs forced us to generalize the model to support it.

3.1. Failure Automaton State

We begin by specifying the types of components of a failure automaton state.

[Event]

An event represents either a *basic event* or a *derived event* in the system. A basic event represents an external phenomenon, or a component failure. A derived event represents the composition of several other events, and often corresponds to a subsystem.

$StateOfEvents == Event \rightarrow \mathbb{N}$

$StateOfEvents$ is a total function from $Event$ to \mathbb{N} . The function represents the number of operational replicates for each event. If the high-level language does not support replication then the number of replicates is one.

$History == \{ h : \text{iseq } StateOfEvents \mid h \neq \langle \rangle \wedge (\forall i, j : \text{dom } h \bullet \text{dom}(hi) = \text{dom}(hj)) \bullet h \}$

For high-level modeling languages with order-dependent semantics, a history is required in order to record the order in which events occur in the system. A $History$ is specified as a non-repeating sequence of $StateOfEvents$ that represents the changing state of the fault tree after a number of transitions in the failure automaton. The condition states that the history is non-empty, and that every step in the history has the same set of events (although the event states can change).

$SpareInUse == \{ siu : Event \leftrightarrow Event \mid siu \in \mathbb{F}(Event \times Event) \bullet siu \}$

We declare $SpareInUse$ as a partial function from $Event$ to $Event$. The condition ensures that the function is finite. The domain represents a subset of the subsystems in the model which utilize spares, and the range is the spare currently being used. $SpareInUse$ is a partial function because not all subsystems utilize sparing, and a given subsystem may not have any spare which can be used.

$SystemStatus ::= Operational \mid FailedCovered \mid FailedUncovered$

It is possible for a basic event failure to occur in two ways, *covered* and *uncovered*. An uncovered failure means that the failure of a basic event caused a catastrophic system failure. In this case, the system status will be $FailedUncovered$. A covered failure means the basic event failed normally, and the effect on the system is defined by the fault tree. In this case, the system status will either be $Operational$ or $FailedCovered$, depending on the status of the system-level event in the fault tree.

$FailureAutomatonState$ $stateOfEvents : StateOfEvents$ $history : History$ $spareInUse : SpareInUse$ $spares : \mathbb{F} Event$ $systemStatus : SystemStatus$ <hr/> $stateOfEvents = last history$
--

This schema defines the type $FailureAutomatonState$. A failure automaton state consists of the current state of the events, the history, the allocation of spares, the set of basic events which are spares, and the system status. The invariant states that $stateOfEvents$ must be equal to the last state of events in the history.

3.2. Failure Automaton Transitions

$FailureAutomatonTransition$ $from : FailureAutomatonState$ $to : FailureAutomatonState$ $causalBasicEvent : Event$ <hr/> $to.history = from.history \hat{\ } (to.stateOfEvents)$ $causalBasicEvent \in \text{dom } from.stateOfEvents$ $from.stateOfEvents causalBasicEvent >$ $to.stateOfEvents causalBasicEvent$ $from.systemStatus = FailedUncovered \Rightarrow$ $to.systemStatus = FailedUncovered$
--

A transition between states consists of a *from* state, a *to* state, and an associated *causal basic event*. We distinguish a particular basic event because high-level languages may support functional dependencies such that the failure of one component causes another to immediately fail as well. In this case, the intermediate language must be able to identify the event which caused the state transition.

The invariants state several constraints on the relationships between the states and the causal basic event. The destination state extends the history of the source state by one set of event states. The causal basic event must be one of the events in the event state, and it must be the case that additional replicate failures of the basic event occur in the transition between states. Finally, a system which has failed uncovered must remain that way.

3.3. Failure Automata

$FailureAutomaton$ $states : \mathbb{F} FailureAutomatonState$ $transitions : \mathbb{F} FailureAutomatonTransition$ <hr/> $states = \bigcup \{ t : transitions \bullet \{ t.from, t.to \} \}$

A failure automaton consists of a finite set of states and transitions. The predicate constrains the transitions to be between the states of the failure automaton.

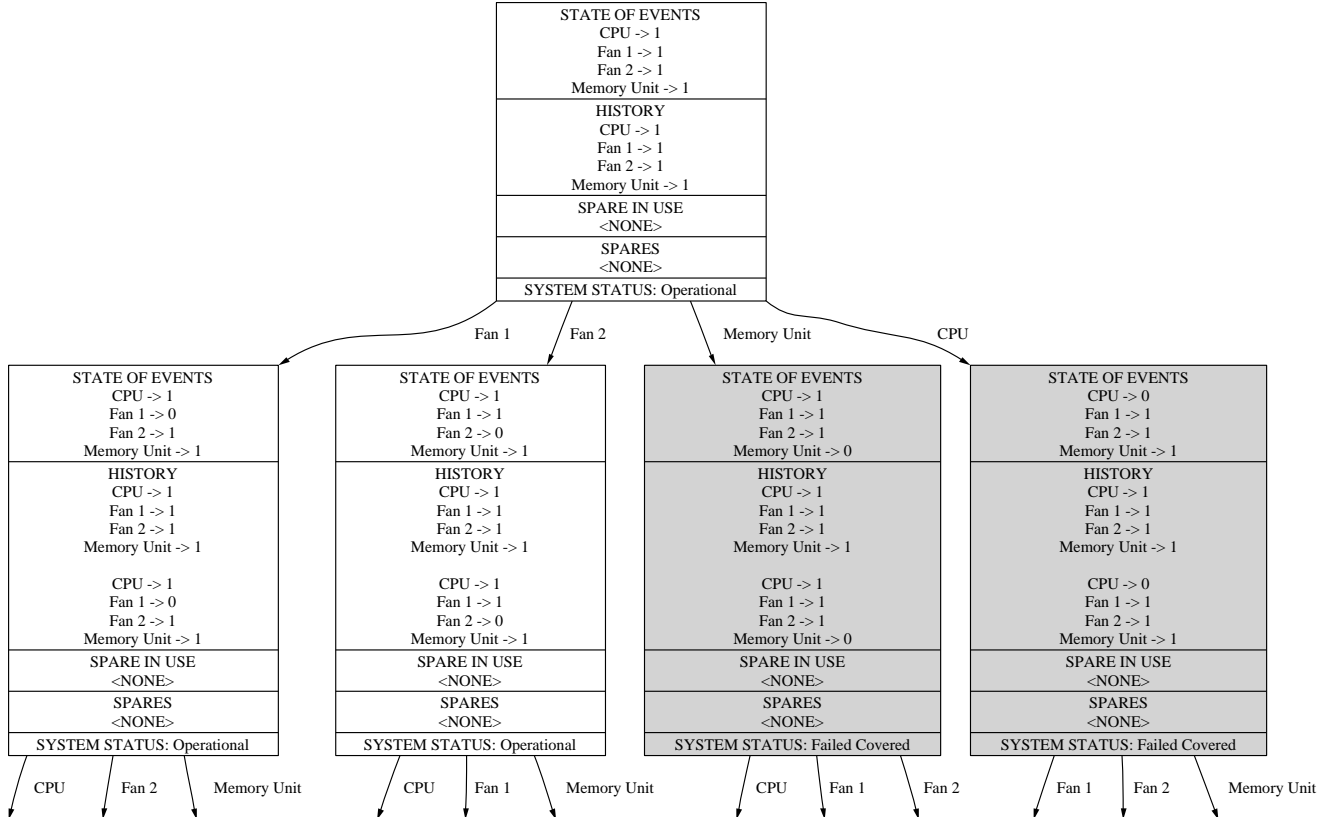


Figure 3. Part of the FA for the example RBD.

4. Semantics of Reliability Block Diagrams

In this section we present a formal definition of RBDs, and their semantics in terms of FAs. Because RBDs are a combinatorial-based language, some of the features of FAs will not be necessary. For example, while each FA state will have a history, it will not play a role in the semantics of RBDs. Similarly, state components related to sparing will not be used either.

Figure 3 shows a portion of the FA for the example RBD shown earlier in Figure 1. In the initial state at the top of the diagram, each component of the system is operational. The history consists of only the current state of events. The system status is operational because there exists a path from source to sink in the RBD. In this state as well as the others, spare allocation and list of spares are both empty because RBDs do not support sparing.

From the initial state, there are four components which can potentially fail. Each of these failures is a causal basic event which results in a unique next state. For the left two states resulting from a failure of one of the fans, the system is still operational because of the fan redundancy. The right two states are both failed covered because the failure of the

memory unit or CPU means that there is no path from the source to sink in the RBD. None of the states in the FA have a failure status of *FailedUncovered* because our definition of RBDs does not support modeling of uncovered failures.

4.1. Reliability Block Diagrams

In this section we specify the RBD. We begin with basic types.

$[Block, Component]$

Here we define given types for the blocks of an RBD and the system components being modeled.

$DestsMap == Block \rightarrow \mathbb{F}_1 Block$

We define the connectivity of blocks in an RBD as a partial function from a block to a nonempty finite set of blocks. For example, in Figure 1, the CPU block's output connects to the inputs of both fan blocks, so there would be a mapping from the CPU block to the set of blocks $\{Fan\ 1, Fan\ 2\}$.

$PhysToLogMap == Component \rightarrow \mathbb{F} Block$

This function defines the relationship between components in the system and blocks in the RBD. We use this function to associate multiple blocks for a given component in the system. When the component fails, all of the associated blocks should also fail.

$$\begin{array}{l} \text{IsDirectlyOutputTo } _ : \mathbb{P}(\text{Block} \times \text{Block} \times \text{DestsMap}) \\ \hline \forall \text{from, to} : \text{Block}; \text{outputs} : \text{DestsMap} \mid \text{from} \in \text{dom outputs} \bullet \\ \text{IsDirectlyOutputTo}(\text{from}, \text{to}, \text{outputs}) \Leftrightarrow \text{to} \in \text{outputs.from} \end{array}$$

This axiomatic function computes a Boolean value indicating whether the *from* block outputs directly to the *to* block, according to the connectivity information stored in the *outputs* mapping.

$$\begin{array}{l} \text{IsOutputTo } _ : \mathbb{P}(\text{Block} \times \text{Block} \times \text{DestsMap}) \\ \hline \forall \text{from, to} : \text{Block}; \text{outputs} : \text{DestsMap} \mid \text{from} \in \text{dom outputs} \bullet \\ \text{IsOutputTo}(\text{from}, \text{to}, \text{outputs}) \Leftrightarrow \\ \text{IsDirectlyOutputTo}(\text{from}, \text{to}, \text{outputs}) \vee \\ (\exists g : \text{dom outputs} \bullet g \in \text{outputs.from} \wedge \text{IsOutputTo}(g, \text{to}, \text{outputs})) \end{array}$$

This axiomatic function computes a Boolean value indicating whether the *from* block outputs directly or indirectly to the *to* block, according to the connectivity information stored in the *outputs* mapping.

$$\begin{array}{l} \text{RBD} \\ \hline \text{sourceBlock} : \text{Block} \\ \text{sinkBlock} : \text{Block} \\ \text{reliabilityBlocks} : \mathbb{F} \text{Block} \\ \text{components} : \mathbb{F} \text{Component} \\ \hline \text{dests} : \text{DestsMap} \\ \text{physToLogMap} : \text{PhysToLogMap} \\ \hline \text{sourceBlock} \notin \text{reliabilityBlocks} \\ \text{sinkBlock} \notin \text{reliabilityBlocks} \\ \text{sourceBlock} \neq \text{sinkBlock} \\ \hline \text{dom dests} = \{\text{sourceBlock}\} \cup \text{reliabilityBlocks} \\ \forall b : \text{dom dests} \bullet \text{dests } b \subseteq \{\text{sinkBlock}\} \cup \text{reliabilityBlocks} \\ \forall b : \{\text{sinkBlock}\} \cup \text{reliabilityBlocks} \bullet (\exists c : \text{dom dests} \bullet b \in \text{dests } c) \\ \forall b : \text{dom dests} \bullet \neg \text{IsOutputTo}(b, b, \text{dests}) \\ \hline \forall c_1, c_2 : \text{dom physToLogMap} \mid c_1 \neq c_2 \bullet \\ \text{physToLogMap } c_1 \cap \text{physToLogMap } c_2 = \emptyset \\ \bigcup \{c : \text{dom physToLogMap} \bullet \text{physToLogMap } c\} = \text{reliabilityBlocks} \end{array}$$

An RBD comprises a single source block and sink block, a finite set of reliability blocks, a finite set of system components, a function defining the connectivity of the blocks, and a function defining the relationship between the components and blocks.

The invariants define the relationships between state components. The source and sink blocks must be unique. Only the source block and reliability blocks can have output blocks. Only the reliability blocks and sink block can have input blocks. Every block which can be an output is an output for some block. There are no cycles in the outputs. The last two constraints ensure that the blocks associated

with the components partition the set of reliability blocks.

4.2. Semantics of RBDs in terms of FAs

We now present the semantics of RBDs in terms of FAs. We define a number of *Semantics* functions which are true if the RBD semantics are correct with respect to the particular issue that the function addresses. The overall semantics is then defined as a function which maps an RBD to an FA, such that all of the *Semantics* functions are true.

$$\text{ComponentEventCorr} == \text{Component} \mapsto \text{Event}$$

This type is a correspondence between components in the RBD and events in the FA. A component is mapped to the event that represents its failure.

$$\begin{array}{l} \text{GetFAEvents} : \text{FailureAutomaton} \rightarrow \mathbb{F} \text{Event} \\ \hline \forall \text{fa} : \text{FailureAutomaton}; \text{es} : \mathbb{F} \text{Event} \mid \\ (\exists \text{fas} : \text{fa.states} \bullet \text{es} = \text{dom fas.stateOfEvents}) \bullet \\ \text{GetFAEvents}(\text{fa}) = \text{es} \end{array}$$

This helper function extracts the events in a given FA.

$$\begin{array}{l} \text{RBDComponentsMapToFAEvents } _ : \mathbb{P}(\text{RBD} \times \text{FailureAutomaton}) \\ \hline \forall \text{rbd} : \text{RBD}; \text{fa} : \text{FailureAutomaton} \bullet \\ \text{RBDComponentsMapToFAEvents}(\text{rbd}, \text{fa}) \Leftrightarrow \\ (\exists c2e : \text{ComponentEventCorr} \bullet \\ \text{dom } c2e = \text{rbd.components} \wedge \text{ran } c2e = \text{GetFAEvents}(\text{fa})) \end{array}$$

This Boolean function determines whether there exists a correspondence between the components in an RBD and the events in an FA.

$$\begin{array}{l} \text{GetComponentEventMapping} : \text{RBD} \times \text{FailureAutomaton} \leftrightarrow \\ \text{ComponentEventCorr} \\ \hline \text{dom GetComponentEventMapping} = \\ \{\text{rbd} : \text{RBD}; \text{fa} : \text{FailureAutomaton} \mid \\ (\exists c2e : \text{ComponentEventCorr} \bullet \\ \text{RBDComponentsMapToFAEvents}(\text{rbd}, \text{fa})) \bullet (\text{rbd}, \text{fa})\} \\ \forall \text{rbd} : \text{RBD}; \text{fa} : \text{FailureAutomaton}; c2e : \text{ComponentEventCorr} \mid \\ (\text{rbd}, \text{fa}) \in \text{dom GetComponentEventMapping} \wedge \\ \text{dom } c2e = \text{rbd.components} \wedge \\ \text{ran } c2e = \text{GetFAEvents}(\text{fa}) \bullet \\ \text{GetComponentEventMapping}(\text{rbd}, \text{fa}) = c2e \end{array}$$

In contrast to the previous function, this function computes the correspondence between the components in an RBD and the events in an FA. The function is partial because such a correspondence may not actually exist.

$$\begin{array}{l} \text{SpareSemantics } _ : \mathbb{P}(\text{RBD} \times \text{FailureAutomaton}) \\ \hline \forall \text{rbd} : \text{RBD}; \text{fa} : \text{FailureAutomaton} \mid \\ \text{RBDComponentsMapToFAEvents}(\text{rbd}, \text{fa}) \bullet \\ \text{SpareSemantics}(\text{rbd}, \text{fa}) \Leftrightarrow \\ (\forall \text{fas} : \text{FailureAutomatonState} \mid \text{fas} \in \text{fa.states} \bullet \\ \text{dom fas.spareInUse} = \emptyset \wedge \text{fas.spare} = \emptyset) \end{array}$$

RBDs do not support sparing, so the spare allocation in every state of the failure automaton is empty, as is the set of spare events.

CausalBasicEventSemantics $_$: $\mathbb{P}(RBD \times FailureAutomaton)$

$\forall rbd : RBD; fa : FailureAutomaton;$
 $c2e : ComponentEventCorr \mid$
 $RBDComponentsMapToFAEvents(rbd,fa) \wedge$
 $c2e = GetComponentEventMapping(rbd,fa) \bullet$
 $CausalBasicEventSemantics(rbd,fa) \Leftrightarrow$
 $(\forall fat : fa.transitions \bullet$
 $fat.causalBasicEvent \in \text{ran } c2e \wedge$
 $(\forall fas : FailureAutomatonState; c : Component \mid$
 $fas \in fa.states \wedge c \in rbd.components \wedge$
 $fas.stateOfEvents(c2e(c)) > 0 \bullet$
 $(\exists fat : fa.transitions \bullet fat.from = fas \wedge$
 $fat.causalBasicEvent = c2e(c)))$

This Boolean function is true if two conditions hold. First, each FA transition’s causal basic event has an associated component in the RBD. Second, for every FA state, there must be a transition from that state for every operational component in the RBD. This function ensures that the FA will have a valid structure with respect to component failures in the RBD.

PathExists $_$: $\mathbb{P}(Block \times Block \times RBD \times FailureAutomatonState \times FailureAutomaton)$

$\forall from, to : Block; rbd : RBD; fas : FailureAutomatonState;$
 $fa : FailureAutomaton; c2e : ComponentEventCorr \mid$
 $RBDComponentsMapToFAEvents(rbd,fa) \wedge$
 $(from \in rbd.reliabilityBlocks \vee from = rbd.sourceBlock) \wedge$
 $(to \in rbd.reliabilityBlocks \vee to = rbd.sinkBlock) \wedge$
 $c2e = GetComponentEventMapping(rbd,fa) \wedge$
 $(fas \in fa.states) \bullet$
 $PathExists(from, to, rbd, fas, fa) \Leftrightarrow$
 $(IsDirectlyOutputTo(from, to, rbd.dests) \vee$
 $(\exists b : Block; c : Component \mid b \in rbd.physToLogMap(c) \bullet$
 $IsDirectlyOutputTo(from, b, rbd.dests) \wedge$
 $fas.stateOfEvents(c2e(c)) \neq 0 \wedge$
 $PathExists(b, to, rbd, fas, fa)))$

This function is true if there exists a path from the *from* block to the *to* block. We use the failure automaton state to determine which paths are broken as a result of component failures.

FailureSemantics $_$: $\mathbb{P}(RBD \times FailureAutomaton)$

$\forall rbd : RBD; fa : FailureAutomaton; c2e : ComponentEventCorr \mid$
 $RBDComponentsMapToFAEvents(rbd,fa) \wedge$
 $c2e = GetComponentEventMapping(rbd,fa) \bullet$
 $FailureSemantics(rbd,fa) \Leftrightarrow$
 $(\forall fas : FailureAutomatonState \mid fas \in fa.states \bullet$
 $PathExists(rbd.sourceBlock, rbd.sinkBlock, rbd, fas, fa) \Rightarrow$
 $fas.systemStatus = Operational \wedge$
 $\neg PathExists(rbd.sourceBlock, rbd.sinkBlock, rbd, fas, fa)$
 $\Rightarrow fas.systemStatus = FailedCovered)$

This function ensures that the system status of a given FA state will be operational if there is a path from the source

block to the sink block, and will be failed covered otherwise. Note that *FailedUncovered* is not a possible system status, because our formulation of RBDs does not support modeling of uncovered failures.

Having specified the conditions for valid semantics, we now define the overall RBD semantics in terms of FAs.

RBDSemantics : $RBD \rightarrow FailureAutomaton$

$\forall rbd : RBD; fa : FailureAutomaton \bullet$
 $RBDSemantics(rbd) = fa \Leftrightarrow$
 $RBDComponentsMapToFAEvents(rbd,fa) \wedge$
 $CausalBasicEventSemantics(rbd,fa) \wedge$
 $SpareSemantics(rbd,fa) \wedge FailureSemantics(rbd,fa)$

The overall semantics is a *total* function which maps any given RBD to an associated FA. The function states that *fa* is the failure automaton for *rbd* if and only if each of the semantic functions previously defined are true.

5. Semantics of Dynamic Fault Trees

In this section, for the sake of completeness, we sketch the semantic mapping of DFTs to FAs. This section makes it clear that the FA provides a unified semantic domain for both RBDs and DFTs. For space reasons we cannot show the entire semantics—interested readers should refer to previous publications [4, 7] for a semantics of DFTs in terms of an earlier version of the FA.

5.1. Overview

Figure 4 shows a portion of the FA for the example DFT presented earlier in Figure 2. The upper state represents the state of the system after the occurrence of *Event A* from the initial state (not shown). In this state, *Event A* is not operational, while the other two basic events, *Event T* and *Event S* are operational, as well as the event associated with *SpareGate*. The history shows a sequence of two sets of event states—the initial “all operational” state and the current one. The spare gate is using *Event S* because its preferred input *Event A* has failed. *Event A* and *Event S* are listed as spares in the state, and the system is operational because the spare gate is operational.

From this state, there are two operational basic events that can fail, *Event T* and *Event S*. Because *Event S* is a replicated basic event, its failure results in a state in which the state of *Event S* is 1 instead of 2. If either basic event fails, the system is still operational because there is an additional replicate of *Event S* which the spare gate can use.

For presentation purposes we have not shown states with *FailedUncovered* failure status. Essentially, each operational basic event in a state results in a FA transition away from that state to a target state (as we have shown), but also

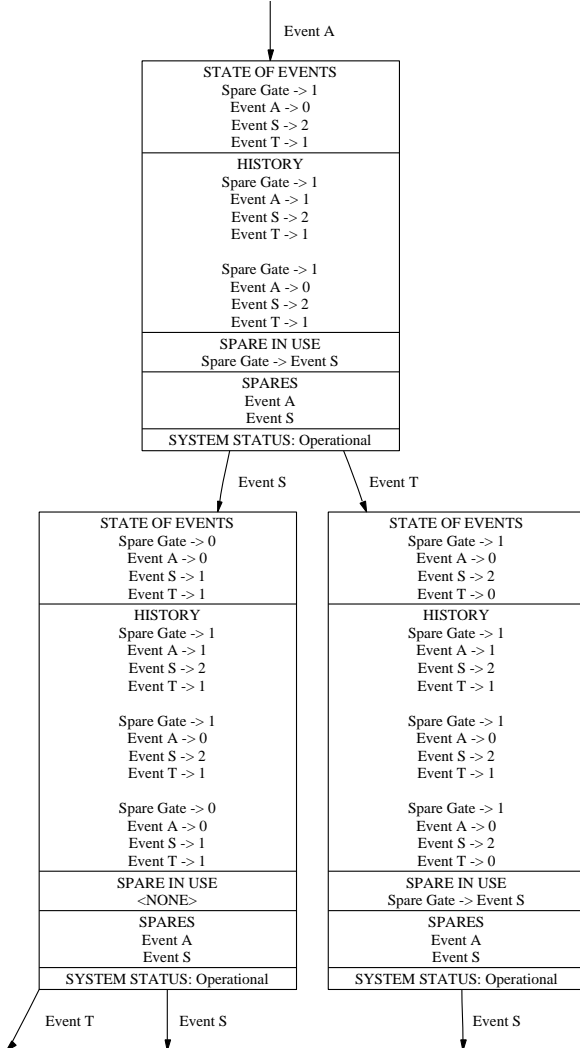


Figure 4. Part of the FA for the example DFT.

results in an additional arc to another target state which is identical except that the system status is failed uncovered.

While we do not present the DFT specification, we describe it briefly. The given type *Event* represents both gates and basic events in the DFT. The basic events of a DFT are specified as a finite set of events, as are the AND, OR, Spare, and other types of gates. Sequence-enforcing constraints which state that events must fail in a particular order are specified as a set of sequences of events. Functional dependencies are relations from a trigger event to a sequence of dependent events. Two partial functions, *replications* and *thresholds*, map events to attributes—*replications* maps basic events to their replication value, and *thresholds* maps threshold gates to their associated threshold values. There is a unique system event in the DFT.

5.2. Semantics of DFTs in terms of FAs

The semantics of RBDs which we presented was fairly straightforward in that the overall system failure depended only on the presence of a path in the RBD given a set of components that have failed. In contrast, DFTs are a complex mixture of different types of constructs, each of which has a different meaning and behavior. In order to better structure the somewhat larger specification, our strategy was to specify the semantics of each type of gate individually, then conjoin them as part of the overall *FaultTreeSemantics* definition. In the interest of space we show only the semantics of AND gates.

$ANDSemantics_{-} : \mathbb{P}(FaultTree \times FailureAutomaton)$

$\forall ft : FaultTree; fa : FailureAutomaton \mid$

$FaultTreeAndFailureAutomatonEventsMatch(ft,fa) \bullet$

$ANDSemantics(ft,fa) \Leftrightarrow$

$(\forall ag : ft.andGates; fas : FailureAutomatonState \mid fas \in fa.states \bullet$
 $NumberOfOccurredReplsInInputs(ft.inputs.ag.fas.stateOfEvents) =$
 $NumberOfReplsInInputs(ft.inputs.ag.ft.replications) \Rightarrow$
 $fas.stateOfEvents.ag = 0 \wedge$
 $NumberOfOccurredReplsInInputs(ft.inputs.ag.fas.stateOfEvents) \neq$
 $NumberOfReplsInInputs(ft.inputs.ag.ft.replications) \Rightarrow$
 $fas.stateOfEvents.ag = 1)$

This Boolean function defines the correct behavior of the AND gates of a DFT. For every DFT and FA where there is a correspondence between the events in the DFT and the events in the FA, $ANDSemantics(ft,fa)$ will be true if and only if, for every AND gate in every FA state, the gate is failed if all of the inputs of the gate are failed, and the gate is operational otherwise.

In a similar way, we define the semantics for each of the different types of gates and constraints. For example, the priority-AND gate specification states that all of the inputs must have occurred, and that they also must have occurred in the specified order.

$FaultTreeSemantics : FaultTree \rightarrow FailureAutomaton$

$\forall ft : FaultTree; fa : FailureAutomaton \bullet$

$FaultTreeSemantics(ft) = fa \Leftrightarrow$

$FaultTreeAndFailureAutomatonEventsMatch(ft,fa) \wedge$
 $CausalBasicEventSemantics(ft,fa) \wedge$
 $SystemEventSemantics(ft,fa) \wedge$
 $ANDSemantics(ft,fa) \wedge ORSemantics(ft,fa) \wedge$
 $ThresholdSemantics(ft,fa) \wedge PANDSemantics(ft,fa) \wedge$
 $SpareGateSemantics(ft,fa) \wedge SEQSemantics(ft,fa) \wedge$
 $FDEPSemantics(ft,fa)$

The overall DFT's semantics is specified in a manner similar to that of RBDs. The complete fault tree semantics, expressed in terms of failure automata, is the conjunction of the causal event, system failure, gate, and constraint semantics.

```

class RBD
{
protected:
    Block source_block;
    Block sink_block;
    set<Block> reliability_blocks;
    set<Component> components;

    Dests_Map dests;
    Phys_To_Log_Map phys_to_log_map;

public:
    // Constructors, etc. omitted

    const bool Block_Has_Outputs(const Block& in_block) const;
    const bool Containing_Component_Exists(
        const Block& in_block) const;
};

```

Figure 5. The RBD class

6. Implementation Experiences

In order to investigate the extent to which the failure automaton can ease the implementation of DFTs and RBDs, we implemented the semantics of both as part of two reliability analysis tools. The first tool, called *Nova*, computes the unreliability of DFTs, and the second tool, *Firefly*, computes the reliability of RBDs. *Nova* was developed as part of an effort [6] to improve the dependability and usability of the Galileo DFT tool [5, 15, 16]. *Firefly* is a novel contribution of this work.

Typically, tools such as these are implemented by translating the high-level domain-specific models into low-level mathematical representations which are then solved using common mathematical libraries. Our goal was to attempt to raise the level of abstraction to the level of the failure automaton, implementing the FA as a library which can be used by solvers for languages such as DFTs and RBDs. We hoped that this experiment would demonstrate that the amount of effort required to implement high-level reliability modeling languages would be greatly reduced.

We had previously developed an implementation of the FA as part of the Galileo project [5, 15, 16]. The FA library as well as the *Nova* and *Firefly* programs are written in C++. The solver converts any given FA into a matrix which encodes a set of differential equations. These equations are then solved using an off-the-shelf FORTRAN solver.

6.1. Implementation for Ease of Verification

Our goal was to develop a trustworthy implementation of the FA solver, so we closely followed the formal specification, creating C++ classes for each of the types present in the Z specification. Similarly, we eschewed performance optimizations in favor of correctness and verifiability. A

Tool	FA	High-Level Semantics	Total
Galileo	N/A	1322	1322
Nova	733	524	1257
Firefly	733	90	823

Figure 6. LOC for semantics in each tool

forthcoming paper is planned on this aspect of our approach. In this paper, we just sketch the basic ideas.

We used our previous FA solver as a library for the implementation of *Nova* and *Firefly*. As in the implementation of the FA solver, we first developed classes corresponding to the types in the formal definitions of the languages. For example, consider the abbreviated definition of the RBD class in Figure 5. Consulting the formal specification of this type in Section 4, one can see that the protected data members of the class exactly mirror those of the specification. Types such as *Dests_Map* and *Phys_To_Log_Map* are defined in the same way, closely following the specification to ease verification that the code matches the specification.

The class has the normal C++ constructors and destructors, as well as operators for comparing two objects of the type, and the usual methods for accessing and modifying the attributes. In addition, the method *Block_Has_Outputs* is used to help guarantee that all reliability blocks and the source block have outputs. The method *Containing_Component_Exists* helps to guarantee the range of *phys_to_log_map* partitions *reliability_blocks*. These are just some of the functions used to satisfy the invariants of the specification.

6.2. Code Comparison

We investigated whether we could save effort in the implementation of the high-level languages by using a reusable library which implements the intermediate language. As a coarse measure of this effort, we counted the number of non-comment, non-whitespace lines of code in each implementation. Figure 6 gives a summary of the lines of code in each tool devoted to the semantics of the high-level model in terms of Markov chains. As a reference we provide corresponding data from Galileo [15], which we had developed without the use of the FA language and implementation.

The “FA” category is the number of lines used to implement the FA abstraction and its semantics in terms of Markov chains. The “High-Level Semantics” category is the lines of code used to implement the translation from the high-level language to the intermediate (*Firefly* and *Nova*) or the lower-level (*Galileo*) representation.

The total lines of code required to implement *Nova* and *Galileo* is approximately the same. However, we were able to structure the *Nova* implementation to use the FA abstrac-

tion, so that less than half of the implementation was for expressing the semantics of DFTs in terms of FAs. In contrast, Galileo uses more code to express the semantics of DFTs because it lacks an intermediate abstraction to help bridge the semantic gap.

To implement Firefly’s RBD semantics, we reused the FA abstraction unchanged. This saved us the effort of re-implementing and re-verifying this aspect of the tool. Because of the simple semantics of RBDs, we were able to implement the semantics using only 90 lines of new code.

While these tools represent only two data points, we believe that the FA abstraction reduced the cost of implementing the semantics of high-level reliability modeling languages. The ease of implementing the RBD semantics suggests that the effort expended is also proportional to the complexity of the high-level language.

7. Evaluation

We believe that our experiences demonstrate that this approach helps to bridge the semantic gap between high-level languages and their low-level semantics, in two primary dimensions. First, the conceptual distance from the high-level language to a well-understood domain is significantly reduced. Because we formally defined the intermediate FA domain, we could more easily define the semantics of DFTs and RBDs. In fact, we were able to *formally* define the semantics of these languages, which is rarely done but very beneficial [7]. Because the FA was formally defined as a Z specification, we were able to express the semantics of DFTs and RBDs more easily. For example, the formal definition of RBDs took approximately a week.

Second, the difficulties of implementing the semantics of high-level languages are also reduced. The FA library reduces the code development effort but more importantly provides a standard, formally-based, debugged codebase for a large fraction of the implementation. This helps to reduce the chance of implementation error. As an example, we have found only one error in the Firefly implementation. While this anecdote does not represent an adequate analysis of the reliability of our software, we hope that it is indicative of the benefits of this approach.

8. Related Work

For some time, the reliability community has understood that there are many similarities between different modeling notations. Malhotra and Trivedi [13] characterize the modeling expressibility of multiple reliability languages in terms of a power hierarchy. For example, RBDs and static FTs are equivalent in modeling power, while reliability graphs are more powerful, and continuous-time Markov

chains are yet more powerful. The basis for defining this hierarchy is a set of algorithms for transforming models of one type into models of another. The authors do not provide proofs of the correctness of these algorithms, nor do they attempt to develop an intermediate language which relates the features of the high-level modeling languages.

The concept of intermediate languages is one that has been borrowed from the programming language and compiler communities. For decades compilers have used intermediate languages to abstract hardware-specific details, allowing optimizations and other code transformations to be easily done before generating object code [10]. Similarly, compilers such as the GNU compiler collection [17] have used intermediate representations to ease the translation of multiple languages. For example, GCC supports the C, C++, Objective-C, Fortran, Java, and Ada programming languages. Despite the successes of this approach in the programming languages community, we are unaware of any uses of this approach in the development of engineering modeling languages.

This work is an outgrowth of previous work in the formal definition of modeling languages [4, 6, 7]. In our previous work, we highlighted the need for the mathematically precise definition of modeling languages used in the design of critical systems. At that time, the FA was characterized as a DFT-specific intermediate domain. In this paper, we show that it, in some form, has the potential to serve as a common intermediate language for other high-level models.

9. Conclusion and Future Work

Because modeling languages are used in the design of critical systems, engineers must have confidence in the modeling languages and the tools which implement them [12]. The current method of defining high-level modeling languages in terms of low-level mathematical abstractions increases the likelihood that the meaning of the modeling language will be incorrect, or that the software implementation will contain errors. By introducing an intermediate language for a class of modeling languages, we raise the level of abstraction of the semantic domain, thereby reducing the “semantic gap” and associated costs involved in defining and implementing more trustworthy modeling languages and tools.

We have demonstrated the promise of this approach for the domain of reliability modeling and analysis, and for DFTs, and RBDs, in particular, using the intermediate language of failure automata. Of course, the FA is not a panacea for reliability engineering. For example, it lacks features to support modeling of component repair, which is an essential element of some high-level modeling notations.

A next step is to further generalize the FA language and enhance it support additional necessary concepts such as re-

pair. Work is in progress to further evaluate this approach by using the revised FA to express the semantics of Boolean-driven Markov processes (BDMPs) [2] another dynamic extension of static fault trees. We may also investigate the applicability of this approach in other domains such as safety modeling and analysis. Ensuring the correctness of the implementation remains an unsolved problem. We are currently investigating using our formal specifications as the basis for runtime monitoring of the implementations to help guarantee correctness.

Finally, this approach has another potential benefit which we have yet to investigate: features added to the intermediate language can rapidly be back-ported to multiple modeling languages. For example, our DFTs support modeling of uncovered failures while our RBDs do not. Enabling such a capability in RBDs involves adding approximately 10 lines of code to the RBD semantics module, as the necessary support is already in the FA. These benefits, and the benefits for the definition and implementation of modeling languages, appear to support the claim that such intermediate languages have a role to play in the development of languages and tools for engineering modeling and analysis.

Acknowledgments

This work was supported in part by the National Science Foundation under grant ITR-0086003 and by NASA Langley Research Center under contract NAS1-02076 (Kevin Sullivan, principal investigator).

References

- [1] R. E. Barlow and F. Proschan. *Statistical Theory of Reliability and Life Testing*. Holt, Rinehart, and Winston, New York, 1975.
- [2] M. Bouissou. Boolean logic driven markov processes: A powerful new formalism for specifying and solving very large markov models. In *Proceedings of the 6th International Conference on Probabilistic Safety Assessment and Management*, San Juan, Puerto Rico, USA, 23–28 June 2002.
- [3] M. A. Boyd. *Dynamic Fault Tree Models: Techniques for Analysis of Advanced Fault Tolerant Computer Systems*. PhD thesis, Duke University, Department of Computer Science, Apr. 1991.
- [4] D. Coppit. *Engineering Modeling and Analysis: Sound Methods and Effective Tools*. PhD thesis, The University of Virginia, Charlottesville, Virginia, Jan. 2003. URL: <http://www.cs.wm.edu/~coppit/papers/dissertation.pdf>.
- [5] D. Coppit and K. J. Sullivan. Galileo: A tool built from mass-market applications. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 750–3, Limerick, Ireland, 4–11 June 2000. IEEE.
- [6] D. Coppit and K. J. Sullivan. Sound methods and effective tools for engineering modeling and analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 198–207, Portland, Oregon, 3–10 May 2003. IEEE.
- [7] D. Coppit, K. J. Sullivan, and J. B. Dugan. Formal semantics of models for computational engineering: A case study on dynamic fault trees. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 270–282, San Jose, California, 8–11 Oct. 2000. IEEE.
- [8] S. A. Doyle and J. B. Dugan. Dependability assessment using binary decision diagrams (bdd). In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, pages 249–258, Pasadena, California, 27–30 July 1995.
- [9] J. B. Dugan, S. Bavuso, and M. Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability*, 41(3):363–77, Sept. 1992.
- [10] D. J. Frailey. An intermediate language for source and target independent code optimization. In *Proceedings of the SIGPLAN '79 Symposium on Compiler Construction*, pages 188–200, Denver, Colorado, 6–10 Aug. 1979.
- [11] R. Gulati and J. B. Dugan. A modular approach for analyzing static and dynamic fault trees. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 57–63, Philadelphia, Pennsylvania, 13–16 Jan. 1997.
- [12] J. C. Knight. Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, pages 547–9, Orlando, Florida, 19–25 May 2002. IEEE.
- [13] M. Malhotra and K. S. Trivedi. Power-hierarchy of dependability model types. *IEEE Transactions on Reliability*, 43(3):493–502, Sept. 1994.
- [14] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
- [15] K. J. Sullivan, J. B. Dugan, and D. Coppit. The Galileo fault tree analysis tool. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 232–5, Madison, Wisconsin, 15–18 June 1999. IEEE.
- [16] K. J. Sullivan and J. C. Knight. Experience assessing an architectural approach to large-scale systematic reuse. In *Proceedings of the 18th International Conference on Software Engineering*, pages 220–229, Berlin, Germany, 25–30 Mar. 1996. IEEE.
- [17] The Free Software Foundation. The GCC home page. URL: <http://gcc.gnu.org/>.
- [18] W. E. Vesely, J. Dugan, J. Fragola, J. Minarick III, and J. Railsback. *Fault Tree Handbook with Aerospace Applications*. National Aeronautics and Space Administration, Aug. 2002.
- [19] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. U. S. Nuclear Regulatory Commission, NUREG-0492, Washington DC, 1981.
- [20] H. A. Watson and B. T. Laboratories. Launch control safety study. Technical report, Bell Telephone Laboratories, Murray Hill, NJ, 1961.
- [21] L. Xing and J. B. Dugan. Analysis of generalized phased mission system reliability, performance and sensitivity. *IEEE Transactions on Reliability*, 51(2):199–11, June 2002.