

Formal Specification in Collaborative Design of Critical Software Tools

David Coppit and Kevin J. Sullivan

University of Virginia Department of Computer Science
Thornton Hall, Charlottesville, VA 22903

{coppit|sullivan}@cs.virginia.edu

Abstract

Engineers use software tools to model and analyze designs for critical systems. Because important design decisions are based on tool results, tools must provide valid modeling constructs; engineers must understand them to validate their models; and tools must implement these constructs without significant error. Such tools thus demand careful conceptual and software design. An important aspect of such design is the use of rigorous specification and advanced design techniques. This paper contributes a case study on the use of such techniques in the collaborative development of a fault tree analysis tool. The collaboration involved software engineering researchers knowledgeable about software specification and design and reliability engineering researchers expert in fault tree techniques. Our work revealed conceptual and implementation errors in an earlier version of the tool. Our study supports the position that there is a need for rigorous software specification and design in developing novel analysis tools, and that collaboration between software engineers and domain experts is feasible and profitable.

1. Introduction and motivation

Engineers rely on software tools to model and analyze engineered systems. Such tools aid in design by analyzing the properties of *software models* of such systems. These tools enable more cost-effective design with greater confidence, but they also pose risks. Such tools and the modeling constructs they support are themselves complex software, and as such resist analysis and are susceptible to error. Hatton has documented divergences in the computations performed by purportedly functionally identical seismic analysis tools [7]. The U.S. Nuclear Regulatory Commission has alerted users of tools used in the design of nuclear reactors of significant errors in such tools [15].

Errors in conceptual design, documentation, or implementation can lead to erroneous decisions for critical systems. Expertise in a given analysis domain is not enough to build high assurance software tools.

Because analysis tools are complex software systems, their production requires the use of modern software engineering techniques. In this paper, we present a case study suggesting that formal (mathematical) software specification methods and modern design methods can help by aiding in the production of trustworthy modeling constructs and tool implementations. Beyond the use of well known specification techniques, this paper addresses a separate, socio-technical problem. People who develop tools are expected to be experts in the given engineering analysis domains—e.g., reliability or mechanical engineering; but they cannot be expected to be and often are not software engineers. Conversely, software engineers are adept with techniques for developing complex, computerized, conceptual structures, but they are not expected to be experts in the application domains. Thus, high assurance tools are likely to emerge only from organizations in which domain and software experts work together.

This paper emerges from such a collaboration. We have been working with Dugan at the University of Virginia in developing the Galileo fault tree analysis tool. The specification that we present was developed with Dugan as domain expert and the authors as software experts. It was both possible and profitable to work collaboratively to develop a modest specification of the tool's modeling constructs. The specification revealed certain conceptual and implementation errors in the modeling constructs implemented in an earlier tool, DIFTree [6]. The specification also provides a basis for precise user documentation of the modeling constructs. Though our experience is modest, we find this a promising approach to designing critical tools.

Section 2 of this paper discusses dynamic fault tree analysis and the Galileo project. Section 3 presents background on formal specification using the Z language. Section 4 presents elements of our specification to make our

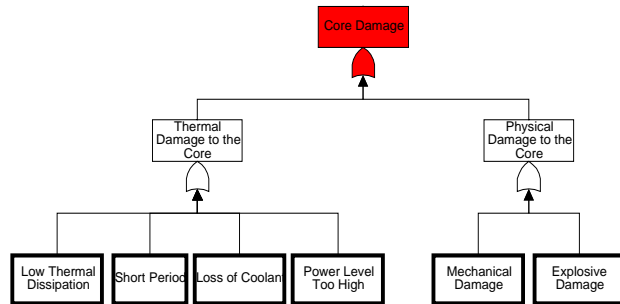


Figure 1. A Simple Static Fault Tree

approach concrete. Section 5 summarizes clarifications and corrections in the conceptual design and implementation that our effort produced. Section 6 evaluates the contribution of this paper. Section 7 concludes.

2. Background

In this section we present an overview of dynamic fault tree analysis [1] and the Galileo fault tree analysis tool.

2.1 Dynamic fault trees

Fault trees [14] were developed to facilitate analysis of the Minuteman missile system [16]. They provide a compact, graphical, easily understood method of to analyze system reliability. Static trees use Boolean gates to represent how component failures combine to produce system failure. Dynamic trees [4,5] add a temporal notion: system failures can depend on the order of component failures.

Figure 1 presents a simple static fault tree. The top-level node, “Core Damage”, an OR gate, models the system as failing if either “Thermal Damage to the Core” or “Physical Damage to the Core” occurs. The leaf nodes, or *basic events*, model the stochastic failure characteristics of basic components. A fault tree tool calculates system failure probabilities. For example, if “Mechanical Damage” and “Explosive Damage” have constant failure probabilities of $\frac{1}{2}$, then the probability “Physical Damage to the Core” is $\frac{3}{4}$.

The static gates are thus defined informally as follows:

- AND. The output failure event occurs if all non-dummy input events occur. (Dummy inputs are from certain gates that have no outputs.)
- OR: Fails if any input fails.
- KOFM: Fails if k of m non-dummy inputs fail.

Dynamic trees extend static trees to enable modeling of time dependent failures. They can model dynamic replacement of failed components from pools of spares; failures that occur only if others occur in certain orders; dependencies that propagate the failure of one component to others; and specification of constraints on failure orders that simplify analysis computations. The dynamic gates are defined informally as follows:

- PAND (Priority AND). Fail if non-dummy inputs failed in given order.
- SEQ (Sequence enforcing): Assert that failures can only occur in given order.
- FDEP (Functional dependency): Propagate failure of trigger input to dependent basic events.
- CSP, WSP, HSP (Cold, warm, and hot spare): When the primary input fails, available spare inputs are used in order until none are left, at which time the gate fails. The “temperature” of the gate selects the degree to which the failure rate of the spares is attenuated.

DIFTree and Galileo solve static trees by converting them to annotated binary decision diagrams [2] and using efficient solution algorithms. Dynamic trees are solved by converting them to continuous time Markov chains [4], and then to differential equations, solved by standard techniques [8]. States correspond to combinations of basic component failures, with each state leading to the next as a result of one additional failure. The transition rate between two states is the failure rate of the newly failed component. States from which no more states are generated (such as system-level failure states) populate the ends of the Markov chain, and are called absorbing states. The chain size depends on the number of basic events and the gate types, with worst case exponential behavior.

2.2 Galileo

This work was done within the Galileo project [9], which is investigating component-based development with massive components. We call our design approach *package-oriented programming* (POP) because it is based on using software packages as components. To evaluate POP, we hypothesized that it would enable rapid, low-cost development of industrially viable engineering tools. Galileo is our experimental vehicle. It uses Microsoft Word, Access, Internet Explorer, and Visio Corporation’s Visio Technical to create a rich tool offering textual and graphical editing of fault trees and many other features. Figure 2 presents a screen shot of Galileo (in an early version). In the upper right is Visio, showing the graphical representation of the fault tree in text form as viewed through Word in the lower right. On the left is Microsoft Access, and in the upper left is the main Galileo window, which coordinates the behavior of the other components. The analysis engine uses a modified DIFTree algorithm.

3. Formal software specification and Z

The software engineering challenge is the cost effective design of complex conceptual structures that address the needs of users. The verified implementation of such structures on real machines, with all of their idiosyncrasies, including finite memories and strange arithmetic, presents additional challenges. The documentation of con-

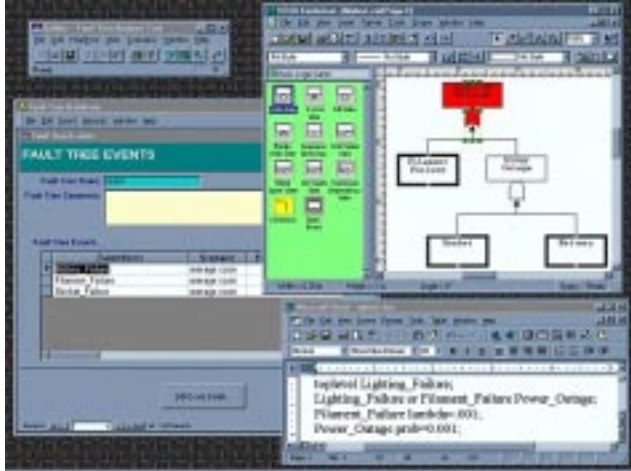


Figure 2. A Screenshot of Galileo

ceptual structures to enable people to use them effectively increases the burden. Finally, the evolution of conceptual structures, implementations, and documentation presents enormous problems.

3.1 Formal software specification

The heart of the matter is the need for precise, abstract, documented specifications of the essential conceptual structure that software systems embody. Such a specification provides a basis for validating a conceptual structure against intuitively understood requirements, for verifying that a computer program implements the structure faithfully, and for user documentation that enables error-free exploitation of the capabilities of the provided program.

In the absence of a documented specification of a software system of significant complexity, the engineer must be concerned for the integrity of its conceptual structure, the faithfulness of a purported implementation, and the accuracy of its documentation. Specifications written in expressive languages with mathematically defined semantics provide a basis for validating and verifying complex conceptual structures and programs that implement them. The process of writing such a specification forces the designer to think carefully, often raising questions that reveal errors in conceptual design. Formal specifications are also subject to analysis, e.g., checking for syntactic consistency, and computer-assisted verification of properties.

On the other hand, comprehensive use of formal methods is infeasible for any but small systems, which rich software tools are not. One pragmatic compromise is the limited use of formal methods to aid in developing critical aspects of such systems. That is the course that we have taken. We present a partial formal specification of key aspects of dynamic fault trees, gates, and basic events.

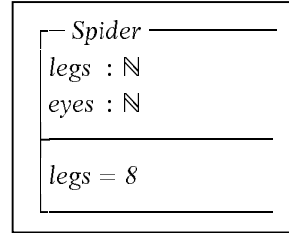


Figure 3. A Simple Schema.

3.2 Formal software specification in Z

To enable specifications expressive enough to be subject to validation against intuitively understood requirements, abstract enough not to unduly constrain design, and mathematically

precise, the Z (“zed”) specification language takes first order logic and monomorphic typed set theory as a basis. Every value has one type. A type is defined by a set disjoint from the sets defining other types. Basic types include \mathbb{N} , the non-negative integers, and \mathbb{N}_1 the positive integers.

Z provides mechanisms to define new types in terms of existing ones. If S is a type, for example, $\mathbb{P} S$ denotes the type *power set of S*. An element of this type is thus a set of elements of S . In addition to logical operators such as *and* (\wedge) and *or* (\vee), and set operators such as intersection (\cap) and union (\cup), Z supports useful relational constructs. For example, a relation between two types is a type whose elements are ordered pairs from the domain and codomain sets; a function is a single-valued relation; an injective sequence is a function representing a sequence of items without duplicates; etc.

A type is used to specify the state space of a system, with each value representing a possible system state. A relational types can specify an operation as a relation on system states respectively before and after the operation is applied. We do not see all of these uses of Z in this paper.

To be understandable a large specification must be decomposed into chunks that are understandable in isolation. The structuring mechanism in Z is the *schema*. A schema defines a Z type, specifying the “shape” of elements of the type. See Figure 3. The declaration section, above the line, declares *state components*, which are something like instance variables of a class in object-oriented design. Each element of the *Spider* type thus has *eyes* and *legs*, whose values are natural numbers. The predicate section, below the line, constrains these values: a spider has eight legs. Z also provides operators constituting a *schema calculus* that provides the means to glue together specification fragments, such as schemas, into specifications. We will see rudimentary use of the schema calculus in this paper.

4. Formal specification of dynamic fault trees

Although static fault trees are well understood, DIFtree dynamic fault trees involve novel and subtle modeling constructs, especially in the time-dependent gates and their interactions. To provide a rigorous basis for the conceptual design of DIFtree, for verifying an implementation, and

BasicEventState	
<i>FaultTreeNode</i>	
<i>distribution</i>	: <i>DistributionType</i>
<i>coveredFailureFactor</i>	: \mathbb{R}
<i>restorationFailureFactor</i>	: \mathbb{R}
<i>singlePointFailureFactor</i>	: \mathbb{R}
<i>replication</i>	: \mathbb{N}_+
<i>numberAvailable</i>	: \mathbb{N}
<i>numberAllocated</i>	: \mathbb{N}
<i>numberFailed</i>	: \mathbb{N}
<i>dormancy</i>	: \mathbb{R}
<i>failTimes</i>	: $\mathbb{F} \mathbb{N}$
# <i>failTimes</i> \leq <i>replication</i>	
$0 \leq$ <i>dormancy</i> ≤ 1	
$0 \leq$ <i>coveredFailureFactor</i> ≤ 1	
$0 \leq$ <i>restorationFailureFactor</i> ≤ 1	
$0 \leq$ <i>singlePointFailureFactor</i> ≤ 1	
<i>coveredFailureFactor</i> + <i>restorationFailureFactor</i> + <i>singlePointFailureFactor</i> = 1	
<i>numberAllocated</i> + <i>numberAvailable</i> + <i>numberFailed</i> = <i>replication</i>	

Figure 4. The Basic Event Specification

for documentation we developed a partial specification in Z. It specifies the gates, how each is evaluated at a given state of the modeled system, and the permitted structure of a fault tree as a composition of basic events and gates. We have not yet specified how dynamic trees are solved to compute system reliability.

In this section we describe our specification and present several of the more important schemas. Space limitations prevent us from presenting the entire specification. The full specification is in preparation for publication as a technical report. The parts that we present build on parts presented in the full document. In particular, we elide our specifications of two important types: *FaultTreeNode* and *DistributionType*. The *FaultTreeNode* type specifies common aspects of basic events and gates, namely that each instance has a unique identifiers. The details are not important here. Throughout this document, we use the shorthand *node* to refer to both gates and basic events.

Probability distributions model the failure characteristics of basic events. If a fault tree is static, the distributions provide a basis for a closed form equation for the probability of failure for the basic events for a given mission time. For dynamic fault trees, the distributions provide transition rates (or *hazard rates*) for a Markov chain. We specified a type called *DistributionType* to model the different kind of supported distributions: constant, exponential, log normal, and Weibull. Each distribution has one or more parameters, with constraints upon their values. For brevity we do not present the details in this paper.

4.1 Formal specification of basic events

Figure 4 presents the specification of a basic event. The inclusion of the *FaultTreeNode* (*FTN*) schema within the

Gate	
<i>FaultTreeNode</i>	
<i>inputs</i>	: \mathbb{F} <i>FaultTreeNodeType</i>
<i>output</i>	: <i>FaultTreeStatus</i>
<i>failTime</i>	: \mathbb{N}
SumNonDummyInputs(<i>inputs</i>) $\neq 0$	

Figure 5. The Gate Specification

BasicEventState (*BES*) schema uses the schema calculus to indicate that a value of type *BES* has the state components and constraints specified in the *FTN* schema, as well as the state components and constraints specified for *BES*. A *BES* value thus also has a probability distribution, which will model component failure characteristics. Exceptional failure conditions are modeled using the restored failure and single point failure factors. The idea is that if a basic event fails at a particular rate, then it will cause the immediate failure of the system with a rate of λ times the single point failure factor. Similarly, the basic event will be recovered from failure at a rate of λ times the restored failure factor. The normal failure case is called covered failure, and the failure rate is λ times the covered failure factor. The restoration, single point failure factor, and covered failure factor are of type *real number*. The constraint part of the schema state that they sum to 1.

As a notational shorthand for fault tree modeling, replication allows multiple identical components to be represented by a single basic event. Because each replicate is identical and anonymous (with one unique identifier for the set of replicates), we specify as additional properties the number of failed replicates and the number of operational replicates in use as spares (allocated to spare gates) in a given state of the system. A multiplicative *dormancy* factor models spares that fail at a reduced rate. Finally, a basic event has a set of fail times, one for each replicate. The size of this bounded by the number of replicates.

4.2 Formal specification of selected gates

Next we present the specifications of the KOFM, functional dependency, and spare gates. To save space, we elide the AND, OR, sequence enforcing, and PAND gates. We also use a number of functions whose specifications are omitted from this paper:

- *GetOutput*: Whether a gate's output value is operational, failed, or dummy.
- *GetNumberOperational*, *GetNumberAvailable*: Respectively the number of operational and available replicates of a basic event.
- *SumNonDummyInputs*: The number of non-dummy inputs of a gate.
- *GetInputs*: The set of *FaultTreeNodes* that are inputs to a gate.
- *GetFailTimes*, *GetFailTime*: The fail times of replicates of a basic event, or of a gate, respectively

-KOFM-	
Gate	
k	: \mathbb{N}_+
m	: \mathbb{N}_+
<hr/>	
$output \neq dummy$	
$k \leq m$	
$m = \text{SumNonDummyInputs}(inputs)$	
<hr/>	
$\text{SumNonDummyInputs}(\{inputNode: inputs \mid$	
$inputNode \in \text{Gates} \wedge \text{GetOutput}(inputNode) = \text{failed}$	
$\vee inputNode \in \text{BasicEvents} \wedge \text{GetNumberOperational}(inputNode) = 0\}$	
$< k$	
$\Leftrightarrow output = \text{operational}$	
$\text{SumNonDummyInputs}(\{inputNode: inputs \mid$	
$inputNode \in \text{Gates} \wedge \text{GetOutput}(inputNode) = \text{failed}$	
$\vee inputNode \in \text{BasicEvents} \wedge \text{GetNumberOperational}(inputNode) = 0\}$	
$\geq k$	
$\Leftrightarrow output = \text{failed}$	

Figure 6. The KOFM Gate Specification

Figure 5 presents the specification of a gate, defining common properties of all gates. Gates are *FaultTreeNodes*. In addition, they have an output, a fail time, and a finite set of inputs. The sum of the non-dummy inputs must be greater than zero.

4.2.1 KOFM. Figure 6 presents the specification of the KOFM gate type. This gate defines a subsystem that fails if k of m non-dummy inputs fail. This type is thus specified as having two components k and m of type positive natural. The output can not be a dummy. k must be less than m . m is the number of non-dummy inputs. The second-to-last predicate states the conditions under which the output is not failed: if and only if the sum of the inputs that are either failed gates or basic events with no operational replicates is less than k . Conversely, the last predicate states that the output is failed if and only if the sum of inputs that are either gates that are failed or basic events with no operational replicates is greater than or equal to k .

4.2.2 Functional Dependency. Figure 7 presents our specification of the functional dependency gate. The output of an FDEP gate is dummy. In addition to the normal characteristics of a gate, FDEP splits its inputs into a trigger and dependent inputs. The trigger can be a gate with an output that is not dummy or a basic event with a replication of one. The dependent inputs must be basic events or gates with dummy outputs. If the trigger is a failed gate or a basic event with no operational replicates, then all the dependent inputs must be failed and have fail times that are less than or equal to that of the trigger. (Dependent inputs can fail prematurely by themselves or as a result of being failed by another functional dependency gate.)

4.2.3 Spare Gate. The last gate in this paper is the spare gate, in Figure 8. The informal DIFtree specification distinguished cold, warm, and hot spare gates, with the temperature determining how to interpret the dormancy factor

-FDEP-	
Gate	
trigger	: <i>FaultTreeNodes</i>
dependentInputs	: \mathbb{F} <i>FaultTreeNodes</i>
<hr/>	
$output = dummy$	
$\langle\langle trigger \rangle, dependentInputs \rangle$ partition $inputs$	
$trigger \in \text{Gates} \wedge \text{GetOutput}(trigger) \neq dummy$	
$\vee trigger \in \text{BasicEvents} \wedge \text{GetReplication}(trigger) = 1$	
$\forall inputNode: dependentInputs \bullet inputNode \in \text{BasicEvents}$	
$\vee inputNode \in \text{Gates} \wedge \text{GetOutput}(inputNode) = dummy$	
<hr/>	
$trigger \in \text{Gates} \wedge \text{GetOutput}(trigger) = \text{failed}$	
$\vee trigger \in \text{BasicEvents} \wedge \text{GetNumberOperational}(trigger) = 0 \Rightarrow$	
$(\forall inputNode: dependentInputs \mid inputNode \in \text{BasicEvents} \bullet$	
$\text{GetNumberOperational}(inputNode) = 0)$	
$\wedge (\forall inputNode: dependentInputs \mid inputNode \in \text{BasicEvents} \bullet$	
$(trigger \in \text{Gates}$	
$\wedge (\forall replicateFailTime: \text{GetFailTimes}(inputNode) \bullet$	
$\text{GetFailTime}(trigger) \geq replicateFailTime))$	
$\vee (trigger \in \text{BasicEvents}$	
$\wedge (\forall replicateFailTime: \text{GetFailTimes}(inputNode) \bullet$	
$replicateFailTime \leq \text{maximum}(\text{GetFailTimes}(trigger))))$	

Figure 7. The FDEP Gate Specification

in input basic event. Writing this specification led to our dropping this distinction. Spare gates of different temperatures could not share spares, as there could be a contradiction in interpreting dormancies. Engineer had to use warm spare gates in cases where different dormancy values were desired for different inputs. Our generic spare gate provides flexibility while allowing cold and hot spare gates to be simulated with dormancy zero or one.

The inputs to a spare gate are partitioned into the primary and spares. An ordering specifies the sequence in

-SpareGate-	
Gate	
primary	: <i>BasicEvents</i>
spares	: \mathbb{F} <i>FaultTreeNodes</i>
inputOrder	: <i>iseq</i> <i>FaultTreeNodes</i>
inputBeingUsed	: <i>BasicEvents</i>
<hr/>	
$output \neq dummy$	
$\text{GetReplication}(primary) = 1$	
$\langle\langle primary \rangle, spares \rangle$ partition $inputs$	
$ran\ inputOrder = inputs$	
$inputBeingUsed \in inputs$	
<hr/>	
$output = \text{operational} \Leftrightarrow \{inputBeingUsed\} \neq \emptyset$	
$output = \text{failed} \Leftrightarrow \{inputBeingUsed\} = \emptyset$	
<hr/>	
$\forall inputNode: spares \bullet inputNode \in \text{BasicEvents}$	
$\vee inputNode \in \text{Gates} \wedge \text{GetOutput}(inputNode) = dummy$	
$(\exists inputNode: inputs \bullet$	
$inputNode \in \text{BasicEvents} \wedge \text{GetNumberAvailable}(inputNode) > 0)$	
$\Rightarrow output = \text{operational}$	
$\{inputBeingUsed\} \neq \emptyset \wedge inputBeingUsed \neq primary \Leftrightarrow$	
$(\forall inputNode: inputs \mid inputOrder \sim (inputNode)$	
$< inputOrder \sim (inputBeingUsed) \bullet$	
$inputNode \in \text{Gates} \wedge \text{GetOutput}(inputNode) = dummy$	
$\vee inputNode \in \text{BasicEvents} \wedge \text{GetNumberAvailable}(inputNode) = 0)$	

Figure 8. The Spare Gate Specification

FaultTree	
$ftNodes$: F FaultTreeNodes
$topLevelGate$: FaultTreeNodes
<hr/>	
$topLevelGate$	$\in ftNodes$
$\forall ftNodeA, ftNodeB: ftNodes \bullet$	$GetFaultTreeNodeID(ftNodeA)$ $= GetFaultTreeNodeID(ftNodeB) \Leftrightarrow ftNodeA = ftNodeB$
$TransitiveReachabilityFrom(topLevelGate)$	$= ftNodes \setminus \{topLevelGate\}$
$GetOutput(topLevelGate)$	$\neq dummy$
$TransitiveReachabilityTo(topLevelGate)$	$= \emptyset$
$\forall ftNodeA, ftNodeB: ftNodes \bullet$	$\neg (ftNodeB \in TransitiveReachabilityFrom(ftNodeA)$ $\wedge ftNodeA \in TransitiveReachabilityFrom(ftNodeB))$
$\forall aNode: ftNodes \mid aNode \in BasicEvents \bullet$	$\# \{ aGate: ftNodes \mid aGate \in SpareGates \wedge GetInputBeingUsed(aGate) = aNode \}$ $\leq GetNumberOperational(aNode)$
$\forall nodeA, nodeB: ftNodes \mid nodeA \in SpareGates \wedge nodeB \in SpareGates \bullet$	$GetPrimary(nodeA) = GetPrimary(nodeB) \Leftrightarrow nodeA = nodeB$

Figure 9. The Valid Fault Tree Specification

which spares are used. The output is not dummy. The primary is a basic event with replication one. One input is designated as the one currently in use. It is set to a special value to indicate that no input is in use if and only if the spare gate is failed.

All inputs are basic events or gates with dummy outputs. If a replicated event input has a replicate available, the gate output is operational. If a spare is in use, all inputs less than it in order must have no replicates available. This condition ensures that spares are used in order, and that all replicates of a spare are used before using the next spare.

4.3 Formal specification of a valid fault tree

We now specify valid dynamic fault trees. See Figure 9. A fault tree comprises a finite set of *FaultTreeNodes*, one the top-level node. Every gate is connected to the tree. All gates are reachable from the top-level node. This requirement leads to the need for dummy outputs: without them, some gates could not be connected. The requirement is meant to improve graphical renderings of trees. The specification stipulates that no two spare gates share the same primary. A basic event cannot be allocated to multiple spares at once. Finally, node identifiers are unique: no single node appears multiple times in a tree.

We use several relations whose definitions are elided. The names suggest their meanings. The first predicate states that the top-level node is a node of the tree. The second asserts that nodes are unique, specifying that for any two that they have the same identifier if and only if they are the same node. The third predicate states that other nodes are reachable from the top-level node, which both ensures connectedness of the tree, and that the node

designated as the top-level is top-most. *TransitiveReachabilityFrom* defines the set of nodes that can be reached by recursively traversing the inputs from a given node. A valid top-level node can not have a dummy output, and no other gate can use the top-level node as an input.

In our definition, a valid tree has no cycles. Thus, for any two nodes, it can not be that the first is reachable from the second, and the second from the first. In practice, situations arise in which cycles have to be modeled; but we have not formalized trees supporting this concept.

The # operator denotes the number of elements in a set. The penultimate predicate thus says that for any basic event, the number of spare gates using it is less than or equal to the number of replicates. The final predicate disallows sharing of primaries for spare gates by stating that for all pairs of nodes that are spare gates, the primaries are the same if and only if the nodes are the same.

5. Results

The direct result of this work is a rigorous, though partial, specification of dynamic fault trees, providing a basis for partially validating the DIFtree method, verifying an implementation, and writing precise user documentation. We have already provided feedback to the domain expert, Dugan. While writing the specification, questions were asked of Dugan. A session was held near the end of the specification effort to present the specification. Dugan was able to understand and comment on the specification, despite having no prior experience with Z. As a result, several clarifications and modifications were made to the specification. We did no automated analysis, but merely writing the specification revealed inconsistencies and ambiguities. For example, specifying input failure order for the PAND gate forced us to decide whether simultaneous failures caused by an FDEP could fail a PAND. We also found four implementation errors in the earlier implementation of DIFtree by comparing its behavior to our specification. The next two sections discuss the four specification and four implementation issues in more detail.

5.1 Clarifications to the pre-existing specification

We discovered several significant omissions, ambiguities, or errors in the previous tacit specification.

- The specification was incomplete in that it did not define whether or not a SEQ gate precluded simultaneous events caused by triggering of an FDEP gate.
- The concept of cold, warm and hot spare gates turned out to be problematical. The intention was to permit sharing of spares among spare gates, but sharing of a spare by different spare gates could lead to an inconsistent interpretation of the dormancy factor. When we presented our specification and analysis of this inconsistency to Dugan, she suggested that the three

spare gate types be removed, and that the generic spare gate, equivalent to the warm spare, be used instead. Doing this removed the problem, while simplifying the specification and, we believe, the engineer's understanding of the spare gate.

- Because basic events can be replicated, we found that they cannot be treated as just operational or failed. The interpretation of a replicated event depends on the gate to which it is attached. For example, when attached to an OR gate, it is interpreted as operational if any replicate is operational; but for, an AND gate, as operational only if all replicates are operational

5.2 Corrections to the earlier implementation

The original DIFtree was implemented on UNIX. Most of the cases that arose from our specification effort worked correctly, but a few did not. Writing the specification led to consideration of cases that turned out not to have been handled correctly by the earlier implementation.

- **Cascaded FDEPs were not handled correctly.** If an FDEP fails an event that triggers another FDEP, the second FDEP should fail its dependent inputs. The original implementation assumed that functional dependency gates did not interact, and evaluated each only once. If the second was evaluated before the first, it would not fail its dependent inputs even though the trigger is failed.
- **PAND gates were not evaluated consistently in the face of simultaneous failures of inputs.** When a functional dependency gate simultaneously failed the inputs to a priority and, PAND gates were not consistently evaluated as being operational.
- **Under contention for a spare, the allocation of the spare was implementation-dependent.** The user was not notified of cases where multiple spare gates can contend for available spares owing to simultaneous failures of primaries due to functional dependencies.
- **Sharing of primaries to spare gates was permitted in some cases.** Such sharing should be disallowed as per the designer's intent that a spare gate should have a unique primary component.

6. Evaluation

The contribution of this paper is two-fold: one part technical, and one methodological. In the technical dimension, we have presented a partial formal specification of dynamic fault trees in the DIFtree style, giving precise, mathematical meaning to the basic conceptual structure. Writing the specification led to the discovery of some difficulties in the earlier design and tool implementation. This work led to the refinement of both the conceptual modeling constructs and the tool implementation.

The specification itself hardly represents a major application of formal methods. We have not fully specified the semantics of the analysis, nor have we used formal proofs in the validation of the requirements or in the verification of our implementation, for example. Moreover, as it stands, the specification is not ideal for such purposes. It merges concerns that are better separated. For example, there is no explicit model of the system state for which gates are evaluated. Separating system state from gate evaluation and making both explicit will better reveal the conceptual structure of dynamic fault tree analysis, providing a firmer basis for reasoning about such analysis. Nevertheless, even the modest use of rigorous software engineering methods, focused on making requirements precise in a documented albeit partial formal specification, led to significant refinements in an important reliability engineering analysis approach, and to the identification of significant errors in a supporting software tool.

Thus, perhaps the greater contribution is in having added a data point supporting the need for methodological improvements, and in a case study of such an improvement. We argued that software tools for engineering modeling and analysis applications are complex software systems that implement complex computer-oriented conceptual structures. Both the conceptual structures and supporting software tools implementations should be treated as essentially software, and should be engineered as such.

We presented a case study suggesting that it is both possible and promising to bring application domain experts (e.g., in reliability engineering) together with experts in rigorous software engineering methods to build engineering modeling and analysis tools with sound engineering foundations. Although our work is a single data point, it supports the claim that software engineers can profitably bring formal specification to bear in collaborations with domain experts in designing critical analysis tools.

7. Conclusion and future work

We presented a formal specification of key aspects of dynamic fault trees in the style of DIFtree. The specification provides a mathematically sound basis for reasoning about the design of the underlying conceptual modeling constructs. It also provides an unambiguous document against which an implementation can be verified, whether formally or not. Finally, it provides unambiguous definitions that can serve as a basis for user documentation. Developing the specification led us to recognize conceptual and implementation errors in an earlier version of DIFtree.

We are now developing a new implementation of the analysis code. We will incorporate the changes suggested by the specification. Likewise, we are implementing a new version of Galileo. It will have enhanced validity checks for fault trees as reflected in the specification.

Our work has three beneficiaries. The domain expert benefits from the rigorous scrutiny of the conceptual model. The implementors benefit by being able to refer to an unambiguous document during implementation and verification. Finally, the users of a tool that implements some aspect of the model benefit from more accurate documentation derived from the formal specification.

We have several options for future work. The first is to restructure the specification for improved separation of concerns. We would undertake such a restructuring prior to extending the specification to define the computation of system reliability based on fault tree models. A change that would simplify the specification would be to remove the concept of dummy outputs, leaving that concept as a detail in the graphical layout module. Another possible change would add the concept of the *failure history* of the system modeled and the notion that a dynamic fault tree is evaluated against such a history to determine if it leads to a system-level failure. This approach would make the dynamic aspect of the analysis explicit. Another modification would be to specify fault trees with replication separately from those without, and to specify how trees with replication map to those without. Such a structure might remove the need to treat basic events as a special case, simplifying the specification significantly.

Using formal methods to help validate the specification is another potential area of future work. Proofs of important properties of the specification could, in principle, help the designer to build confidence in the claim that the specification captures the intuitively desired meanings. Formal verification of properties of an implementation against such a specification could similarly build confidence in satisfaction of the specification by the implementation. Our work provides a concrete starting point for exploring this idea. It is unclear at this time that formal validation and verification would be as profitable a use of resources as merely crafting a precise specification.

Acknowledgements

This work was supported in part by the National Science Foundation under grants CCR-9502029 (CAREER) and CCR-9506779. We thank John Knight for discussions in which he emphasized the need for sound engineering of software tools. We thank Joanne Bechta Dugan and Ragan Manian, our fault tree analysis and DIFtree experts.

References

- [1] Mark A. Boyd, *Dynamic Fault tree models: Techniques for Analysis of Advanced Fault Tolerant Computer Systems*, Ph.D. thesis, Department of Computer Science, Duke University, 1990.
- [2] Randal E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, c-35(8):677-691, August 1986.
- [3] Stacy A. Doyle and Joanne Bechta Dugan, "Dependability assessment using binary decision diagrams," In *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing*, FTCS-25, June 1995.
- [4] Joanne Bechta Dugan, Salvatore Bavuso, and Mark Boyd, "Fault trees and Markov models for reliability analysis of fault tolerant systems," *Reliability Engineering and System Safety*, 39:291-307, 1993.
- [5] Joanne Bechta Dugan, Salvatore J. Bavuso and Mark A. Boyd, "Dynamic fault tree models for fault tolerant computer systems," *IEEE Transactions on Reliability*, Volume 41, Number 3, pages 363-377, September 1992.
- [6] Rohit Gulati and Joanne Bechta Dugan, "A modular approach for analyzing static and dynamic fault trees," in *Proceedings of the Reliability and Maintainability Symposium*, January 1997.
- [7] Hatton, L. and A. Roberts, "How accurate is scientific software?," *IEEE Transactions on Software Engineering*, vol. 20 no. 10, pp. 785—97, Oct. 1994.
- [8] L.F. Shampine and H.A. Watts, "Global error estimation for ordinary differential equations," *ACM Transactions on Mathematics and Software*, June 1976, pages 172-186.
- [9] Kevin J. Sullivan, Jake Cockrell, Shengtong Zhang, and David Coppit, "Package-oriented programming of engineering tools," In *Proceedings of the 19th International Conference on Software Engineering*, pages 616-617, Boston, Massachusetts, 17-23 May 1997, IEEE.
- [10] Kevin J. Sullivan, "Galileo: An advanced fault tree analysis tool," [URL:http://www.cs.virginia.edu/~free/index.html](http://www.cs.virginia.edu/~free/index.html)
- [11] Kevin J. Sullivan, "Better, Cheaper, Faster Tools," *Proceedings of the 1997 Reliability and Maintainability Symposium*, Philadelphia, PA., January 13—17, 1997.
- [12] K.J. Sullivan and J.C. Knight, "Building Programs from Massive Components," in *Proceedings of the 21st Annual Software Engineering Workshop*, Greenbelt, MD, Dec. 4—5, 1996.
- [13] K.J. Sullivan and J.C. Knight, "Experience Assessing an Architectural Approach to Large-Scale, Systematic Reuse," *Proceedings of the 18th International Conference on Software Engineering*, Berlin, March 1996, pp. 220—229.
- [14] United States Nuclear Regulatory Commission, *Fault Tree Handbook*, NUREG-0492, 1981.
- [15] United States Nuclear Regulatory Commission, "NRC information notice 96-29: Requirements in 10 CFR part 21 for reporting and evaluating software errors," May 20, 1996.
- [16] H.A. Watson and Bell Telephone Laboratories, "Launch Control Safety Study," Bell Telephone Laboratories, Murray Hill, NJ USA, 1961.