

Developing a High-Quality Software Tool for Fault Tree Analysis

Joanne Bechta Dugan

University of Virginia

*Department of Electrical Engineering
Thornton Hall, Charlottesville, VA 22903*

jbd@Virginia.edu

Kevin J. Sullivan and David Coppit

University of Virginia

*Department of Computer Science
Thornton Hall, Charlottesville, VA 22903*

sullivan@virginia.edu, david@coppit.org

Abstract

Sophisticated dependability analysis techniques are being developed in academia and research labs, but few have gained wide acceptance in industry. To be valuable, such techniques must be supported by usable, dependable software tools. We present our approach to addressing these issues in developing a dynamic fault tree analysis tool called Galileo. Galileo is designed to support efficient system-level analysis by automatically decomposing fault trees into modules that are solved separately using appropriate techniques. Usability is addressed by a software architecture based on a component-based design technique that we call package-oriented programming. We integrate multiple, volume-priced, mass-market software packages to provide the bulk of the tool superstructure. To address tool dependability, we are developing natural language and partial formal specifications of fault tree elements, and we exploit the inherent redundancy associated with multiple analysis techniques as an aid in testing.

1. Introduction

Analysts concerned with quantitative assessment of reliability or safety of fault tolerant computer systems have a variety of mathematical techniques at their disposal: for example, fault trees, Markov and other stochastic processes, and simulation. Each technique has advantages and disadvantages, and the attributes of the system under analysis tend to determine which technique to use. However, it is seldom the case that a single technique is applicable to an entire system, both because of the size of the system and because of the varying attributes of the subsystems. A good reliability engineer thus needs to use different techniques to analyze different parts of a system, decomposing a large complex model into smaller pieces and applying different techniques to each submodel. Most of these techniques are already supported by software tools, but it can be tedious and error-prone to manually decompose a system-level model into submodels, apply

different analysis tools to different submodels, and integrate the results. Thus, a system-level dependability analysis tool must support the integration of several different analysis techniques.

Dependability analysis tools must not only support advanced analytical techniques, but they must do so with a level of sophistication that users now demand of practical software tools. They must support rich functionality such as graphics, persistent storage, report generation, data management, etc. The tools must have a level of usability best acquired through serious usability engineering and conformance to standard user interface conventions. They should run on industrial computing platforms of choice—today Windows-based PC workstations. They must be interoperable, i.e., integrate cleanly with other software systems on those platforms, and with the broader engineering activities of an engineering enterprise. Finally, because the markets for such tools are small, they have to be developed at low cost to avoid prohibitive pricing.

In addition to providing analytical sophistication and usability, users require some assurance that the models they build are valid and interpreted correctly, and that the results that are produced are correct. Thus, the modeling constructs must be precisely defined, so that the analyst has confidence that the model faithfully represents the system, and to provide a sound basis for software design and implementation. The validity of models and analytic results is especially important if a tool will be used to develop mission- or safety-critical applications.

In this paper we present our approach to developing Galileo, a high-quality tool for dynamic fault tree analysis, which addresses these concerns. First, our analysis methodology, called *DIFTree* [11][15], uses a modular approach that combines several different analysis techniques automatically. Second, the architecture of our software tool uses a component-based software development approach that we call *package-oriented programming*. In this style, a few large-scale, widely used, volume priced software packages provide the vast bulk of the

non-analysis functions at low cost while meeting the critical functional, usability and interoperability requirements for sophisticated tools [22][23][24]. Third, we are developing a combination of natural-language and partial formal specifications for the fault tree gates and their interactions to help to validate the modeling framework, to aid users in building valid models and to provide a basis for verifying the implementation of the analysis approach. Fourth, we are exploiting the inherent redundancy associated with multiple analysis techniques as an aid in testing.

The rest of this paper is organized as follows. Section 2 provides background on dynamic fault tree analysis, software engineering, package-oriented programming and Galileo. Section 3 discusses the use of natural-language and formal specifications for fault tree gates. Section 4 characterizes the use of various fault tree analysis techniques for solution and for testing. Section 5 concludes.

2. Background

2.1 Dynamic fault tree analysis

Fault trees [25] were developed to facilitate unreliability analysis of the Minuteman missile system [27]. They provide a compact, graphical, intuitive method to analyze system reliability. Traditional fault trees use Boolean gates to represent how component failures combine to produce system failures, and they are analyzed using cut sets (or other Boolean algebraic methods) or Monte Carlo simulation.

Markov models gradually replaced fault trees as the methodology of choice for reliability analysis of fault tolerant systems after the concept of coverage was introduced and its importance was noted. Coverage modeling can be easily incorporated into Markov models and, until recently, was thought to be difficult to incorporate into fault tree analysis. Further, the complex redundancy management techniques typically used in fault tolerant computer systems (for example prioritized use of spares) can not be captured in combinatorial models like fault trees or reliability block diagrams. However, they can be incorporated easily in state-based models. The analysis methodology of choice for fault tolerant computer systems is thus frequently based on Markov models. Fault trees remain a popular modeling choice for reliability analysis of non-fault tolerant systems. Most reliability engineers are well-versed in fault tree analysis.

Recent work in dynamic fault trees has addressed both limitations and has resulted in a fault tree analysis approach that is applicable to fault tolerant computer systems and non-fault tolerant systems as well. *Dynamic* fault trees add a sequential notion to the traditional fault tree approach: system failures can depend on component failure order as well as combination. Special purpose dy-

namic fault tree gates can model dynamic replacement of failed components from pools of spares, failures that occur only if others occur in certain orders, dependencies that propagate failure in one component to others, and situations where failures can occur only in a predefined order. Fault trees with dynamic gates are typically solved by automatic conversion to equivalent Markov models [10][11].

Traditional (now called static) fault trees have also benefited from recent research. The use of Binary Decision Diagrams (BDDs) has facilitated the solution of very large static fault trees. Some authors have solved fault trees with 2^{10} basic events [8]. A technique for incorporating coverage modeling into a BDD-based fault tree solution was presented in [9]. That work showed that the need for coverage modeling does not necessarily demand a Markov model. Thus, for systems that exhibit no sequence-dependent failure behavior, static fault trees can again be used. The BDD-based approach is much faster than the Markov chain conversion, and yet can easily incorporate the important notion of imperfect coverage.

Researchers have also been exploring the use of divide-and-conquer approaches for analyzing fault trees [6][13][21], since solution time is exponential in the worst-case. Of particular interest is a recently published linear-time algorithm by Dutuit and Rauzy for finding independent subtrees [13]. The algorithm identifies independent subtrees (subtrees which share no basic events) during a depth-first traversal of the tree, recording the first and last visit to each node. This recent development provides the structure needed to combine different solution techniques automatically, as well as providing a means for developing independent Markov models in a dynamic fault tree. Using Rauzy's algorithm [13] on the fault tree model, we can automatically detect independent subtrees, classify them as static or dynamic, and solve them using the most appropriate method. Even if there are only dynamic subtrees, the automatic identification of independent submodels can be of enormous benefit. Compare the solution of three separate Markov models each of 1000 states with the solution of the combined model, containing a cross-product of each state space, and thus a billion states. Further, the use of a fault tree model as the overall system model facilitates the automatic combination of the results of the solution of the submodels.

The *DIFTree* dynamic fault tree analysis methodology first described [15] is a hybrid technique that supports automatic decomposition, analysis, and integration of partial results. During traversal, a subtree is marked as dynamic if a dynamic gate is present. If a subtree contains no dynamic gates, it is classified as static. After the traversal is completed, static subtrees are solved using BDD-based method. The Markov method is used for dynamic subtrees. Our approach fully supports coverage modeling in static and dynamic subtrees. Failure probabilities in

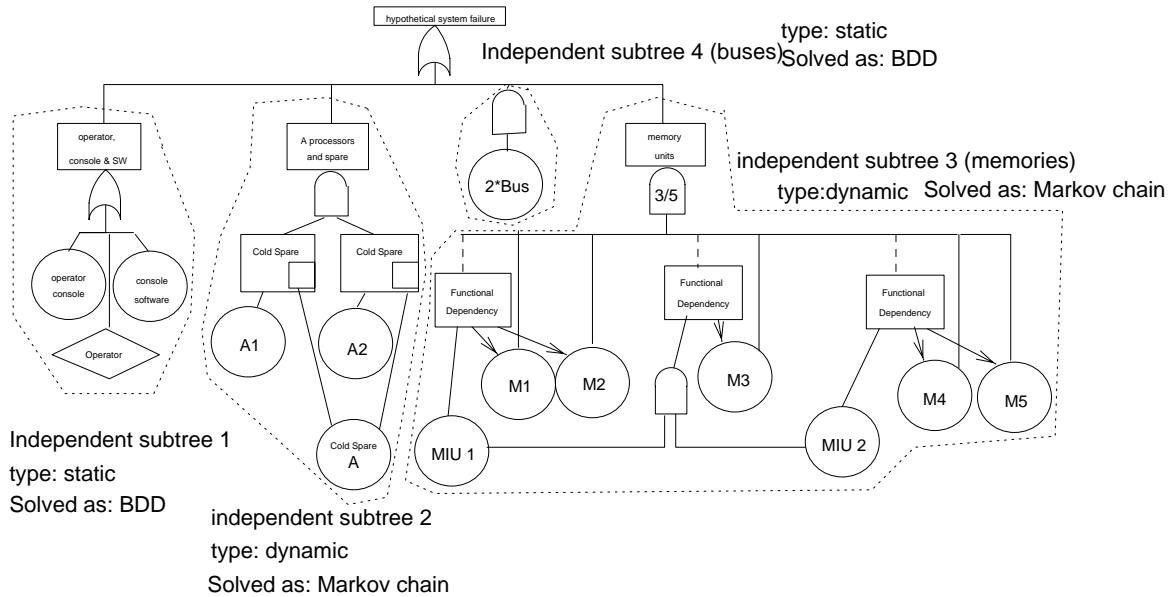


Figure 1. An example modularization

static subtrees may be constant (time-independent) or follow the exponential distribution. Dynamic trees support only the exponential distribution of time to failure.

Figure 1 illustrates the modularization operation of *DIFTree* on a hypothetical fault tree containing two static subtrees and two dynamic subtrees. The static subtrees are solved by automatic conversion to the equivalent BDD, while the dynamic subtrees are solved by automatic conversion to the equivalent Markov model. Each submodel is solved for the probabilities of covered and uncovered failure, and is replaced by a basic event in the higher-level mode. A basic event is characterized by a failure probability and a coverage factor. The reduced, top-level fault tree is then solved as a static tree with four basic events, one representing each subtree. This example was described in more detail in [12].

A static subtree can be recursively split into smaller subtrees without loss of accuracy. That is, the divide and conquer approach to static fault trees does produce an exact solution. However, the further splitting of dynamic subtrees can lead to inaccuracies. The dynamic subtree requires a Markov solution, which in turn depends on an exponential time to failure. Since the time to absorption in a Markov model is not necessarily exponentially distributed, we cannot provide an exact solution if we subdivide a dynamic subtree. Thus, to avoid the use of an unbounded approximation, *DIFTree* does not split dynamic subtrees. We have clearly made a choice of accuracy over performance, as the further splitting of a dynamic subtree may substantially improve solution time. Anand & Soman have presented a similar technique which does split dynamic subtrees, trading accuracy for performance [2].

2.2 Software engineering of modeling tools

Algorithmic advances in engineering modeling and analysis have little chance of having an impact on practice unless they are supported by sophisticated software tools. Unfortunately, such tools are large and complex software systems, often involving a million or more lines of code. These systems are also subject to demanding usability, interoperability and dependability requirements. Specifying, designing, implementing, verifying, correcting and enhancing them requires substantial software engineering expertise and, often, substantial investments in software.

Software engineering difficulties present major impediments both to the dissemination of algorithmic advances, and—perhaps even more seriously—to their effective evaluation and evolution over time based on feedback from use in practice. Researchers who focus on modeling and analysis frameworks generally lack the software engineering knowledge needed to produce software well. Research prototypes tend to be useful as proofs of concept and throw-away prototypes but not as practical tools. Furthermore, markets for such tools are generally small; yet production costs are high. Thus such tools usually cannot be priced attractively, inhibiting commercial production, as well. The need for such tools is unmet by the software engineering state of the art.

The strength of our approach is in the collaboration between fault tree domain experts and software engineering researchers. The core of our attack is based our work on component-based software design using mass-market software packages as components.

The approach, which we discuss in detail in the following subsection, reduces the size of the software problem to the point that it becomes feasible for a small team of capable software engineers to handle; but it is hardly a panacea. A serious challenge remains in the formalization and validation of the modeling and analysis framework. Considerable skill is also needed to specify, design, implement, verify, document, correct and enhance the software not addressed by components: namely, the core modeling and analysis code, and the code to specialize and integrate the component packages with each other and with the core analysis code.

2.3 Package-oriented programming

Package-oriented programming (POP) [22][23][24] is a software development approach in which multiple commercial off the shelf software packages are used as components. By using commercial packages as massive components, POP exploits the vast investments that have already been spent in their design, construction, and refinement and the tremendous economies obtained by the volume pricing of mass-market software. In particular, users benefit from careful usability engineering, rich functionality, software familiarity, rich interoperability, and reasonably stable execution for the level of complexity, at very low cost.

We have shown that the POP approach is particularly appropriate for building software tools. Tools consist of algorithmic analysis cores implemented in perhaps a few tens of thousands of lines of code. However, useful tools also have a “superstructure” supporting such features as textual and graphical interfaces, report generation, etc., the implementation of which can not be done using an amount of effort comparable to that required for the analysis cores. POP addresses this problem through the reuse of packages such as Microsoft Word and Visio Corporation’s technical drawing program to dramatically lower the cost to develop and to use a tool which achieving a high degree of usability and interoperability.

By using the approach to build Galileo we avoided designing a tremendous amount of software from scratch. Instead, we only designed and implemented a fault tree data type and underlying analysis techniques; we specialized the packages for our purposes; and we wrote code to drive the packages and to “glue” them together. In all we have built fewer than 30,000 lines of code, a reduction of several orders of magnitude compared to a build-from-scratch approach producing a comparably useful result.

2.4 Difficulties in component software design

Achieving such benefits through component-based design of complex software in any general sense—whether

using commercial packages or other elements as components—remains a demanding challenge at the forefront of software engineering research. Enabling designers to avoid coding from scratch by using commercial components has been a research goal for decades. Yet, with few exceptions, success has been elusive. Function libraries work, but they address only small aspects of applications. Operating systems and databases have succeeded as massive components, but they only provide infrastructure, not central functions at the application level. Object-oriented programming was once seen as the key but is now widely recognized as not having fostered a component industry.

Indeed, component-based development is increasingly seen as a chimera. In a widely cited paper, Garlan and his colleagues documented a set of severe difficulties encountered in an attempt to integrate a set of large, ostensibly reusable software systems to produce a tool not unlike the one that we are developing. On that basis they concluded large-scale component integration faced fundamental difficulties. In particular, their components made conflicting assumptions about the architectures of the systems in which they would be used, making integrating them very hard [14]. Garlan et al. coined the phrase *architectural mismatch* to describe this kind of problem.

More recently, in a keynote address at the 1999 International Conference on Software Engineering, Butler Lampson argued that the component dream was unlikely to be realized for three reasons: components make conflicting assumptions; they are costly to develop; and they costly to understand [17]. Lampson further argued that the only components that were likely to succeed outside of narrow domains were large, general components: operating systems, databases and web browsers, in particular.

Our work on package-oriented programming is an attempt to thread the eye of this needle. We agree that success requires the use of large components: only they provide adequate design leverage. On the other hand, we hypothesize that components smaller and less general than operating systems, databases and web browsers can succeed: namely shrink-wrapped packages. Such components are costly to develop, but they have the advantage of being sold not only in the component marketplace, but also as end-user applications. Thus, they are inexpensive to *buy* because they are volume-priced. They are also easy to understand for users of system into which they are incorporated because they are popular and well documented. We have not found them easy to work with as designers because of the undocumented and quirky behaviors of their virtual machine (developer) interfaces. Clarifying our knowledge of this and related integration issues is a key aspect of our work on package-oriented programming.

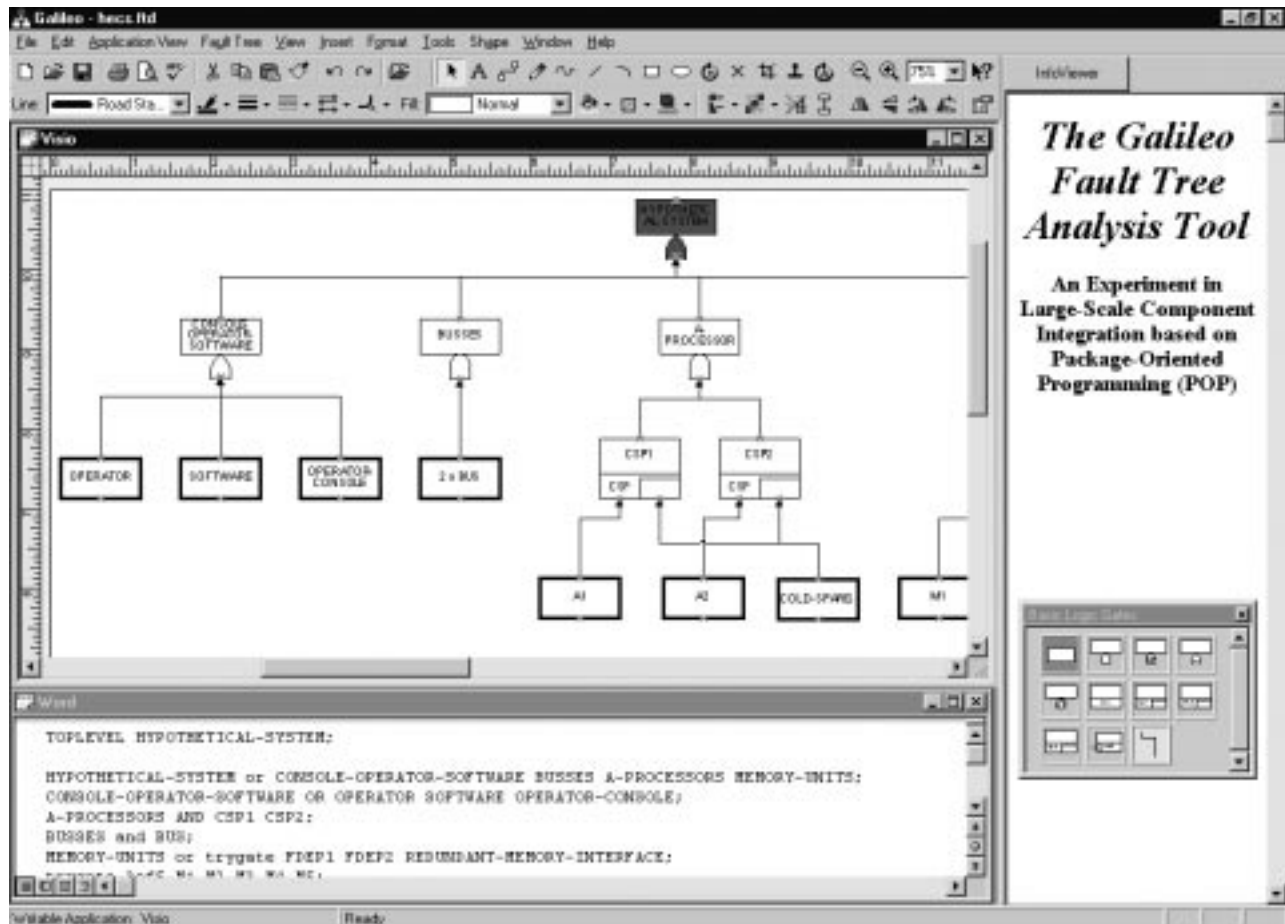


Figure 2. A screenshot of Galileo

At this point, the question arises: Do the difficulties that we encountered rise to the level of the problems of architectural mismatch that Garlan et al. observed in attempting to integrate large components? Our answer is a qualified no. A central theme of our work is that the integration of independently designed components can, and indeed can only, be enabled by conformance to shared design rules, or *integration architectures* [23][26]. We undertook the work reported here knowing that the components that we are using all conform to a common integration architecture, Microsoft's *Active Document Architecture* (ADA) [19].

The ADA in turn is based on lower level architectures: *ActiveX* [5] and ultimately on the *Component Object Model*, COM [20]. Conformance to the ADA suffices to enable (among other things) the integration of the windows presented by separate packages, such as Word and Visio, within a container window, such as that presented to the user of Galileo. Switching among sub-windows, management of menus associated with separate windows, and other such issues are handled automatically. Our components do not exhibit strong architectural mismatch.

Rather the difficulties we have experienced are of other kinds. First, the virtual machines presented by the packages were not always what we needed to implement the functions that we wanted. Second, in some cases, the packages have not fully conformed to the ADA, or they presented surprises, such as the inability to support all virtual machine functions when used in the ADA context.

Despite the problems we have experienced, the approach appears to have considerable potential to enable significant advances in the production of complex systems in some important domains, especially that of engineering modeling and analysis tools. The approach addresses Lampson's concern for development cost by using volume-priced packages as components. It addresses component understanding costs borne by the end user by using familiar packages as elements of the user interface. The problem that *developers* face in using such components remains a serious issue for at least two reasons. One is that the specifications of the exposed virtual machines are not well documented. Another is that the components evolve continually as versions are released. Of course, such evolution is double-edged: it presents prob-

Table 1: Summary of subtree characteristics and solution methods

| | | | | | | |
|-----------------------|-----------------------------|---------------------------------------|-------------------|-------------------|--------------|--------------|
| Parameters | Has Constant Probability? | <i>don't care</i> | Yes | No | No | No |
| | Static or Dynamic Tree? | Static | Dynamic | Dynamic | Dynamic | Dynamic |
| | Uses a Weibull Distribution | <i>don't care</i> | <i>don't care</i> | No | Yes | Yes |
| | Has Cold/ Warm Spare Gates? | N/A | <i>don't care</i> | <i>don't care</i> | No | Yes |
| Analytical Techniques | Cut Sets | Possible | Not Allowed | Not Possible | Not Possible | Not Possible |
| | Binary Decision Diagrams | Preferred | Not Allowed | Not Possible | Not Possible | Not Possible |
| | Markov Chains | Possible if No Constant Probabilities | Not Allowed | Preferred | Possible | Infeasible |
| | Monte Carlo Simulation | Possible | Not Allowed | Possible | Possible | Preferred |

lems, but also presents the opportunity to give major new capabilities to end users at extremely low cost. Finally, our approach addresses architectural mismatch by appealing to the capability of shared integration architectures to enable the integration of independently developed components with certain defined cost and performance properties.

2.5 The user view of Galileo

Figure 2 presents the user's view of the Galileo fault tree analysis tool [25]. In the upper left is a graphical representation of the fault tree whose textual form is viewed in the lower left. The window on the right is used to display documentation and to contact the tool authors. Each of these views is integrated into the main Galileo window.

Figure 2 shows how our architecture uses Visio Corporation's Visio Technical, Microsoft Word, and Microsoft Internet Explorer to create the superstructure needed by fault tree analysis tools. Users benefit from the tremendous investment in the design and implementation of these components. For example, Word supports find-and-replace; Visio, panning, zooming and cut-and-paste of graphical views, etc. Visio also allows the user to manipulate the graphical representation of the fault tree, and then print it or embed it in other documents.

Galileo supports two views of fault trees. The traditional, graphical view allows the engineer to create a fault tree using shapes for the various gates, and connectors that model the relationships between gates. The benefit of this view is that its graphical nature makes it easy to com-

prehend, although it is not as easy to edit for some people as the textual view. The textual view describes the same fault tree using a simple language. The benefit of this approach is easier and faster editing of the fault tree. The drawback is more difficult comprehension.

Each of the packages we used was specialized for use in our POP-based architecture. For example, we customized Visio by creating a "stencil" of fault tree shapes and connectors, and by changing the behavior of mouse clicks to cause the display of information related to each shape for fault tree editing purposes. None of the packages can be "exited" by the user independently of the overall tool. Internet Explorer was programmed to display Galileo documentation.

In addition to restricting the components for better integration into our architecture, we added functionality at the user level that integrates the views. We used Microsoft's Active Document approach to containing multiple documents (a Visio drawing, a Word document, and an Explorer browser) in an overall "container" window. The container ensures that the menus of the currently active document are displayed, and that Galileo-specific menu choices are merged into the menus of the packages. The Galileo menus allow the user to propagate automatically changes made in the textual view to the graphical view and vice-versa. The fault tree menu allows the user to indicate that the fault tree representation currently being edited is to be solved by the analysis engine.

3. Specifying dynamic behavior

Dynamic fault trees [11] augment the standard combinatorial (*AND*, *OR* and *M-out-of-N*) gates with a special set of dynamic gates to model sequential dependency. The original set of four dynamic gates (Functional Dependency, Priority-AND, Sequence Enforcing and Cold Spare) has been expanded to include three more (Hot Spare, Warm Spare and Probabilistic Dependency). The Functional Dependency gate (FDEP) and Probabilistic Dependency (PDEP) gates are used to model (deterministic and probabilistic, respectively) cascading (or common-cause) failures. The Priority-AND (PAND) and Sequence-Enforcing (SEQ) gates are used to detect or prevent certain sequences of events. The spare gates are used to model spare configurations, especially pooled or priority-based spares or those that have a different failure rate when dormant than when active.

The use of dynamic gates has greatly expanded the class of systems to which fault tree analysis can be applied, since the sequential behavior characteristic of fault tolerant computer systems can be effectively captured in a dynamic fault tree. Dynamic fault trees are solved by automatic conversion to the equivalent Markov model [10]. However, using dynamic fault trees in an industrial setting raised two concerns. First, how can an analyst be confident that a model is an accurate representation of the system being analyzed? Second, how can she be confident that the solution is accurate?

Although static fault trees are reasonably well understood, dynamic fault trees involve new and subtle conceptual modeling constructs that are thus subject to error, as well as demanding implementation issues. The semantics of the time-dependent fault tree gates and the interactions between them, in particular, are subtle and subject to misunderstanding. In order to provide a rigorous engineering basis for debugging the conceptual design, for verifying an implementation, and for producing user documentation we developed a partial formal specification of dynamic fault trees in the Z language. It contains formal specifications of static and dynamic gates, how each is evaluated at a given system state, and the permitted structure of a dynamic fault tree as a composition of basic events and gates [7]. In addition to providing a rigorously defined starting point for a redesigned software tool, the specifications helped us to detect and resolve several ambiguities in the gate interactions.

However, the formal specification of gates does not necessarily help a reliability engineer gain confidence that the model being built is an accurate representation of the system under study. Formal specifications can be difficult to read and understand by someone whose expertise lies in a different domain. For this reason, we also developed a set of carefully worded natural language specifications for each gate, based on the formal specification. These

natural language specifications are more complete and precise than they would have been had they not been preceded by the formal specifications and are useful to reliability engineers building models of complex systems [18].

4. Combining analysis techniques

Our approach to the solution of fault trees automatically decomposes the system level fault tree into modules, which are solved separately. Our modular approach allows different subtrees to be solved by different methods: static subtrees can be solved by conversion to an equivalent BDD, while dynamic subtrees can be solved by conversion to the equivalent Markov chain [15]. Recently we have considered the addition of a third solution alternative, and have experimented with the use of a Monte-Carlo simulation engine (MCI-HARP) [4] that uses variance reduction techniques for the analysis of highly reliable systems. The use of simulation as a third alternative not only increases the analysis capabilities of our methodology, but also offers interesting possibilities in terms of multiple solutions of the same subtree. In this section we discuss the decision criteria for choosing a particular solution algorithm, and discuss how we exploited the alternatives as an aid in testing.

4.1 Choosing appropriate analysis techniques

Table 1 summarizes the applicability of several dynamic fault tree analysis techniques. The upper half of the table shows combinations of four characteristics of subtrees: whether a subtree uses constant failure probabilities (as opposed to a distribution of time to failure), whether it has any dynamic gates, whether it has any cold or warm spare gates and whether it uses a Weibull time-to-failure distribution. The lower half describes the abilities of the solution alternatives.

Traditional cut set approaches to fault tree analysis are only applicable to static fault trees and are generally inferior to the newer BDD based approaches. Markov methods are applicable to dynamic subtrees with exponential and Weibull time to failure distributions, as long as the subtree does not combine a Weibull time-to-failure distribution with a cold or warm spare. Simulation presents a viable alternative to the analytical approaches in several interesting situations. The combination of warm or cold spares and Weibull (or other non-exponential) time to failure distributions defies general-purpose techniques, but could be handled easily via simulation. For static subtrees, the combinatorics of the M-out-of-N gate can overwhelm any Boolean algebraic approach if N is large and the inputs are not statistically identical. One example fault tree from industry used a 4-of-12 gate, where each of the 12 inputs was a 5-of-16 gate; functional dependencies

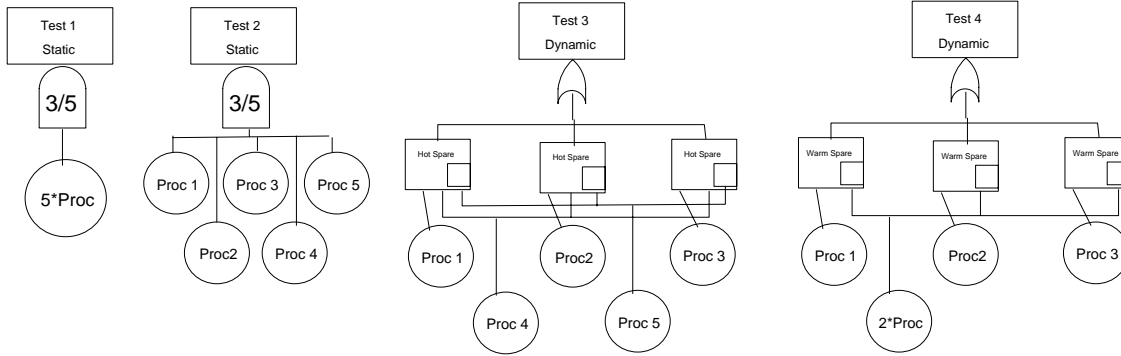


Figure 3. The four fault trees in this figure all produce the same numerical result

required that each input be considered separately. This model could have been analyzed more easily by simulation, especially since the failure probabilities of each basic event were not especially small.

Considering simulation as an alternative solution method poses interesting questions with respect to the preferred method of solution. With only the static (BDD) and dynamic (Markov) classifications, the choice was simple: choose the BDD solution where possible and the Markov solution where necessary. Some combinations (i.e. constant probability of failure in a dynamic model) are disallowed. If we add simulation to the set of solvers, some previously disallowed situations (i.e. cold or warm spares and Weibull time to failure) are now permissible. Further, simulation is applicable to both static and dynamic subtrees and in some cases (i.e. large combinatorics) may be more attractive than the analytical approach.

4.2 Solution techniques as an aid in testing

Because we have multiple solution techniques available within a single tool, we can exploit this flexibility in creating test cases, in two different ways. First, for some trees, multiple solution techniques are applicable, although one may be more efficient than the other. For example, a static fault tree can often be solved by conversion to a Markov model (if exponential or Weibull time to failure distributions are used), even though the BDD-based solution is clearly preferred. Because the approaches used in these solutions are fundamentally different (the BDD solution is based in Boolean algebra and the Markov approach is based on differential equations), we have a basis for greater confidence that the results are correct if both solutions produce the same results. Further, some of the structures we use degenerate into other structures (but with different solution paths) for specific sets of parameters. The exponential distribution is a special case of the Weibull and both the cold and hot spares are special cases of the warm spare. In both the distributional and the spares case, the structure of the Markov

chain is different, but the result (probability of failure) should be the same. We have thus created a set of test cases that exploit these similarities.

Second, we can exploit the different solution techniques by creating different fault trees to model the same scenarios, where one might be static and the other dynamic, or one might contain logical redundancies. For example, some hot spare situations can be adequately modeled using static gates and some redundancy management scenarios can be modeled either with the hot spare gate or with the PAND gate.

As an example of the use of the diverse solution methods for testing, consider the four fault trees shown in Figure 3. Test case 1 is a static tree consisting of a simple 3/5 gate with a replicated basic event. The event is used when there are multiple occurrences of statistically identical components that do not need to be distinguished. Test case 2 expands the replicated event into distinct basic events. Test case 3 uses the hot spare gate to model the redundancy more explicitly while Test case 4 uses the warm spare gate with a dormancy factor of one. The dormancy factor (between zero and one) represents the reduction in the failure rate experienced while the spare is dormant; zero corresponds to a cold spare, one to a hot spare. The first two test cases are static models while the second two are dynamic. We can further extend the test set by varying the coverage parameters (perfect vs. imperfect) and by varying the failure distribution: the degenerate case of the Weibull distribution is the exponential..

In addition to using different solvers in Galileo for testing the analysis, we can compare some static test cases against other fault tree solvers. Figure 4 shows a fault tree (reported in more detail in [1]) solved using Galileo and two commercially available packages. From this test case we learned that some commercially available fault tree analysis packages do not necessarily produce correct results. In fact, both commercial packages produced the same incorrect result because they were incapable of recognizing internal (i.e. non-basic) events which fan out (i.e. are used as input to more than one gate).

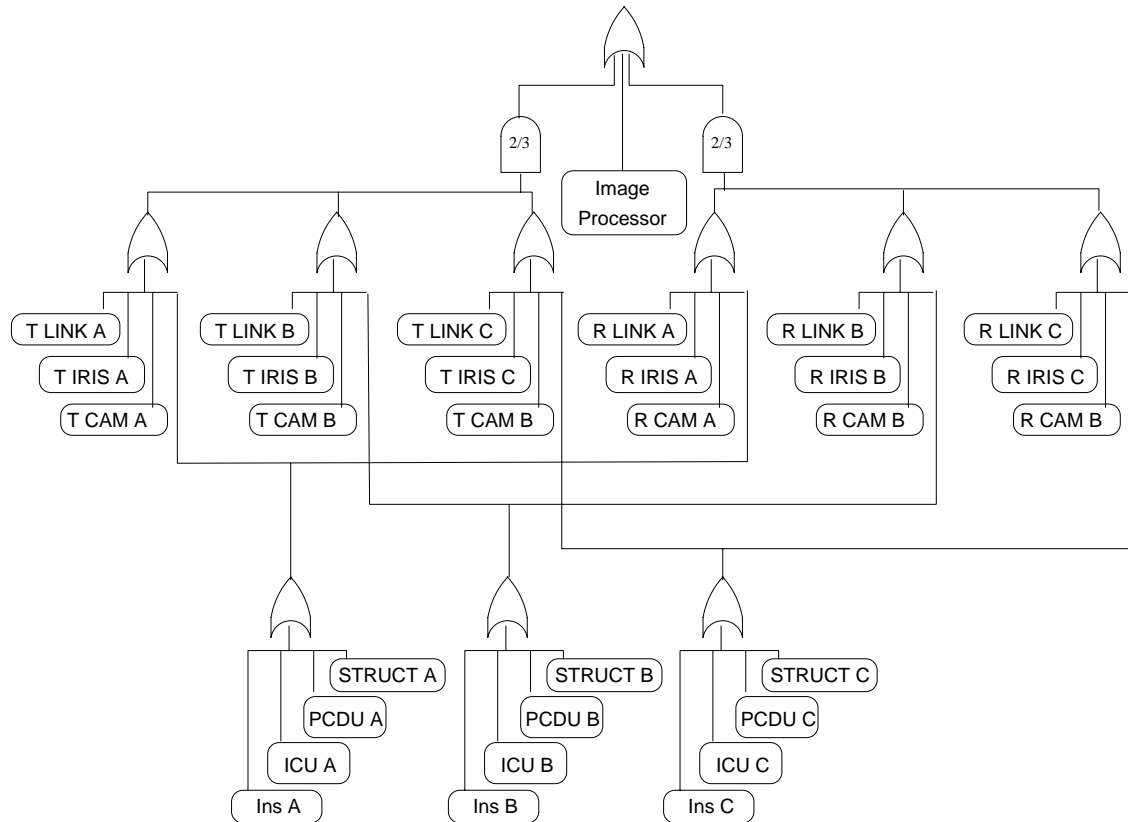


Figure 4. An example fault tree for testing

5. Conclusion

In this paper we presented our approach to the cost-effective development of a dependable and useable software tool for the reliability analysis of complex, fault tolerant computer-based systems. Our approach builds on an effective combination of novel reliability engineering and software engineering methods. We have been concerned not only with achieving a highly usable tool encompassing sophisticated dependability analysis techniques, but also with insuring the fidelity of the analysis. Towards this end, we are applying both formal and natural software engineering specification techniques to clarify the semantics of dynamic fault trees. We have also exploited the design diversity inherent in our analysis approach to implement self-checking for testing purposes.

Our test efforts against commercially available tools for fault tree analysis highlight the need for fidelity in the analysis. Both tools that we used for testing are popular in the reliability engineering community; both tout their ability to provide exact (as contrasted with approximate) solutions, yet both made the same fundamental algorithmic error. We believe that our approach to testing using diverse solution methods will help an analyst gain

confidence in the results from our tool. Instead of always selecting the most efficient solution method, we plan to allow a “solve using all available methods” mode of solution to provide multiple results from diverse solutions.

Acknowledgements

This work was supported in part by the National Science Foundation under grants MIP 95-28258, CCR-9502029 (CAREER), CCR-9506779, CCR-9804078 and by NASA Langley Research Center and Ames Research Center. We thank reliability engineers at NASA Langley Research Center and Lockheed-Martin Corporation for their valuable feedback.

References

- [1] Suprasad Amari, Joanne Bechta Dugan and Ravindra Misra, “A separable method for incorporating imperfect coverage into combinatorial models,” *IEEE Transactions on Reliability* vol. 48, no.3, September 1999.
- [2] Anju Anand and Arun K. Somani, “Hierarchical analysis of fault trees with dependencies, using decomposition,” In

- Proceedings of the 1998 Reliability and Maintainability Symposium*, January 1998, pages 69-75.
- [3] Barry W. Boehm and William L. Scherlis, "Megaprogramming," In *Proceedings of the DARPA Software Technology Conference*, pages 63-82. Meridien Corp., Arlington, VA, 1992.
- [4] Mark Boyd & Salvatore J. Bavuso, "Simulation modeling for long duration spacecraft control systems," *Proceedings of the 1993 Reliability and Maintainability Symposium*, January 1993, pages 106-113.
- [5] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.
- [6] P. Chatterjee, "Modularization of fault trees: A method to reduce cost of analysis," *Reliability and Fault Tree Analysis, SIAM*, 1975, pages 101-137.
- [7] David Coppit and Kevin J. Sullivan, "Formal specification in collaborative design of critical software tools," In *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium*, pages 13-20, Washington, D.C., 13-14 November 1998. IEEE.
- [8] O. Coudert and J.C. Madre, "Fault tree analysis: 10^{20} prime implicants and beyond," In *Proceedings of the Reliability and Maintainability Symposium*, 1993, pages 240-245.
- [9] 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.
- [10] 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*, Volume 39, pages 291-307, 1993.
- [11] 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.
- [12] Joanne Bechta Dugan, Bharath Venkataraman and Rohit Gulati, "DIFTree: A software package for the analysis of dynamic fault tree models," *Proceedings of the 1997 Reliability and Maintainability Symposium*, January 1997, pages 64-70.
- [13] Yves Dutuit and Antoine Rauzy, "A linear-time algorithm to find modules in fault trees," *IEEE Transactions on Reliability*, September 1996.
- [14] David Garlan, Robert Allen, and John Ockerbloom. "Architectural mismatch: Why reuse is so hard." *IEEE Software*, Volume 12, Number 6, pages 17-26, November 1995.
- [15] 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.
- [16] E.J. Henley and H. Kumamoto, *Probabilistic Risk Assessment*, IEEE Press, 1992
- [17] Butler Lampson. "How software components grew up and conquered the world." Keynote address, International Conference on software Engineering. IEEE/ACM, May 1999.
- [18] Ragavan Manian, David Coppit, Kevin J. Sullivan and Joanne Bechta Dugan, "Bridging the gap between systems and dynamic fault tree models," *Proceedings of the 1999 Reliability and Maintainability Symposium*, January 1999, pages 105-111.
- [19] Microsoft. "Active Document Containers." URL: http://msdn.microsoft.com/library/devprods/vs6/visualc/vccore/_core_activex_document_containers.htm
- [20] Dale Rogerson. *Inside COM*. Microsoft Press, 1996.
- [21] A. Rosenthal, "Decomposition methods for fault tree analysis," *IEEE Transactions on Reliability*, Volume 43, June 1980, pages 136-138.
- [22] 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.
- [23] 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, pages 220-229.
- [24] 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.
- [25] Kevin J. Sullivan, Joanne Bechta Dugan and David Coppit, "The Galileo Fault Tree Analysis Tool," *Proceedings of the 29th International Conference on Fault-Tolerant Computing (FTCS-29)*, 1999.
- [26] K.J. Sullivan, M. Marchukov, and J. Socha, "Analysis of a Conflict Between Aggregation and Interface Negotiation in Microsoft's Component Object Model," *IEEE Transactions on Software Engineering*, Sept/Oct 1999, to appear.
- [27] United States Nuclear Regulatory Commission, *Fault Tree Handbook*, NUREG-0492, 1981.
- [28] H.A. Watson and Bell Telephone Laboratories, "Launch Control Safety Study," Bell Telephone Laboratories, Murray Hill, NJ USA, 1961.