Developing a Low-Cost, High-Quality Software Tool for Dynamic Fault Tree Analysis

Joanne Bechta Dugan
Department of Electrical Engineering – University of Virginia

Kevin J. Sullivan
David Coppit
Department of Computer Science – University of Virginia

Key Words —Software Engineering, Software Tools, Reliability Analysis, Fault Tree Analysis, Testing.

Summary & Conclusions — Sophisticated modeling and analysis methods are being developed in academic and industrial research labs for reliability engineering and other domains. The evaluation and evolution of such methods based on use in practice is critical to research progress, but few such methods see significant use. A critical impediment to disseminating new methods is our inability to produce, at a reasonable cost, supporting software tools that have the usability and dependability characteristics that industrial users require and the evolvability to accommodate software change as the underlying analysis methods are refined and enhanced. The difficulty of software development thus emerges as a key impediment to advances in engineering modeling and analysis.

Today, producing sophisticated software tools is costly and difficult even for capable software developers. One problem is that when common design methods, such as object-oriented programming, are used to build such tools, the results are often large, complex, thus costly programs. Tools on the order of a million lines of code are typical, with much of the code devoted to tool interoperability, the human-computer interface and other issues not directly related to modeling and analysis. Making matters worse, domain experts, such as reliability engineering researchers, often lack skills in modern software development, while software engineers and researchers lack knowledge of the application domains. All too often the results of tool development efforts today are thus costly, hard to use, not dependable and essentially unmaintainable.

In this paper, we present an approach to tool development that attacks these problems. Progress requires synergistic, interdisciplinary collaborations between application domain and software engineering researchers. We have pursued such an approach in developing a fault tree modeling and analysis tool called Galileo. We describe our innovations in two dimensions. The first is Galileo's core reliability modeling and analysis function. The second is our work on software engineering for high-quality, low-cost modeling and analysis tools.

In the reliability engineering domain, Galileo supports precise, modular, dynamic fault tree analysis using techniques developed primarily by Dugan and her colleagues. This approach addresses the problem that a single analysis technique is seldom applicable to an entire system. A good reliability engineer uses different techniques to analyze different parts of a system, decomposing a complex model into smaller pieces, applying different analysis techniques to submodels, and integrating partial results into a system-level result. Manually decomposing systems into parts, developing submodels, analyzing them with different tools and techniques, and integrating the partial results is tedious and error prone at best. By contrast, Galileo automatically detects independent sub-trees; translates them into appropriate submodels based on Markov chains, Boolean decision diagrams and other formalisms, analyzes the submodels; and integrates the results. Galileo supports precise analysis while exploiting modularity for scalability in solving problems that require time and space that is exponential in the number of basic events in the worst-case.

Our software engineering approach centers on the component-based design techniques of Sullivan and his colleagues. One key element of the approach is the use of mass-market software packages as large components. We call this *package-oriented programming*. We use it to achieve an effective human-computer interface, tool interoperability, and considerable dependability for the function delivered, at low cost. By low cost, we mean that the effort involves a small handful of graduate and undergraduate students and faculty. We also use Sullivan's mediator-based design approach at several scales to support an integrated, multi-view environment in which it is possible to edit fault trees in either textual or graphical form, while fostering dependability and evolvability. To help us to validate our modeling approach and to verify our implementation, we have used natural language and formal specifications for the fault tree gates and their interactions.

We have evaluated our tool against commercially available fault tree analysis tools. The results highlight the need for fidelity in analysis. Testing two tools popular in the reliability engineering community revealed *the same* algorithmic error in both, despite their claimed ability to provide exact solutions. At the intersection of software and reliability engineering, we exploit the redundancy inherent in the use of multiple analysis techniques in Galileo as an aid to testing our software. Galileo has been acquired by hundreds of sites. We are now building an enhanced version with NASA Langley Research Center.

1. INTRODUCTION

Acronyms

BDD	binary decision diagram
POP	package-oriented programming
FDEP	functional dependency gate
PDEP	probabilistic dependency gate
PAND	priority-and gate
SEQ	sequence enforcing gate

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. These attributes are characteristic of a wide variety of approaches to the modeling and analysis of complex systems.

Software tools that implement these techniques must not only support the underlying mathematical modeling and analysis frameworks, but they must do so with a level of sophistication that users now demand of practical software tools. They must support rich functions such as graphical editing, 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 workstations. They must be interoperable, i.e., integrate cleanly with other software systems typically used 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 [25].

In addition to providing analytical sophistication and usability, users require assurances 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 models faithfully represent systems, 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 [7].

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, we employ a variety of software engineering techniques from our research, the combination of which is intended to deliver the required function and dependability at low cost. 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]. 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. Finally, 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 and the software engineering problems involved in producing tools and ultimately in making advances in engineering modeling and analysis. Section 3 presents the concrete results of our work, the Galileo fault tree tool. Section 4 discusses the central feature of our software engineering approach, namely the use of mass-market software packages as components. Section discusses our use of natural-language and formal specifications to help in the validation of our fault tree modeling framework and in the verification of the Galileo tool. Section 5 characterizes the use of various fault tree analysis techniques for solution and for testing. Section 6 outlines future work.

2. BACKGROUND

2.1 Dynamic fault tree analysis

Fault trees [29] were developed to facilitate unreliability analysis of the Minuteman missile system [30]. 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.

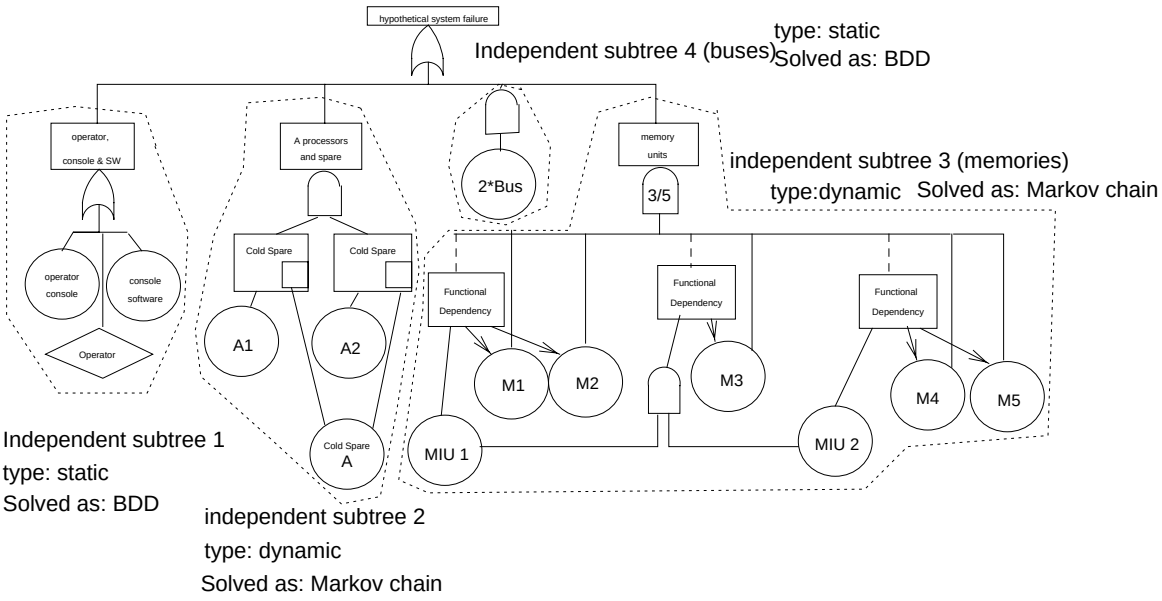

Figure 1. An example modularization

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 dynamic 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 10^{20} 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 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 & Somani 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 being validated effectively or having a significant impact on practice unless they are supported by software tools. Users today demand tools that are quite sophisticated before they consider using them. Unfortunately, such tools are large, complex software systems, often of a million lines of code or more, and subject to demanding functional, usability, interoperability and dependability requirements. Specifying, designing, implementing, verifying, correcting and enhancing them requires software engineering expertise and significant, often prohibitive, investments.

The basic problem is that the state of the art in software engineering does not adequately support cost-effective production of such tools. Difficulties in software development in turn impede to the dissemination of new algorithms, which ultimately discourages their evaluation and evolution through use in practice. Software engineering research on the design of tools for modeling and analysis, combined with research on the underlying modeling and analysis techniques is needed.

The extent of this problem is not fully appreciated today, but it is rapidly becoming apparent. First, application domain researchers who focus on the underlying modeling and analysis frameworks often lack even a basic understanding of modern software engineering methods. The research prototypes that they produce can be useful as proofs of concept and as throw-away prototypes but not as tools that are usable, dependable, interoperable and evolvable enough to be evaluated and refined effectively through significant use in practice.

However, the problem is much deeper than that. Even when capable software engineers work with domain experts to build tools using modern techniques (such as object-oriented design), the results are often extremely costly, hard to use, undependable, hard to change, stovepipe systems. The strength of our research strategy is based on the recognition that we need to combine the best research efforts in modeling and analysis with cutting-edge research in new software development methods to address this problem effectively.

Our attack on the software aspect is based our research on component-based software design using mass-market software packages as components. This approach, which we discuss below, provides the vast bulk of non-analysis software using low-cost, richly functional commercial components, reducing the software that must be built from scratch by orders of magnitude. The research problem is to understand and overcome significant impediments to component-based software design.

Even if successful, using packages is not a panacea. Several major challenges remain. One is the specification and validation of modeling and analysis frameworks, which we address in Section 3. A second challenge is to devise a software architecture for the overall tool. In addition to components, the tool includes modeling and analysis code and code that integrates the component packages with the core analysis code and with each other. We use concepts from Sullivan's mediator design approach extensively in structuring this code for modular development and ease of evolution [27][28]. We address this issue again, but only briefly in the conclusion.

3. THE USER VIEW OF GALILEO

Our primary software artifact is the Galileo fault tree analsyis tool [25]. Figure 2 presents the user's view of Galileo. In the upper left is a graphical representation of the fault tree. The same tree in a domain-specific-language-based textual form appears in the lower left. The window on the right is used to display documentation and to contact the tool authors. Each of the views is integrated into the main Galileo window.

Our tool architecture uses Visio Corporation's Visio Technical, Microsoft Word, and Microsoft Internet Explorer to create much of the human-computer interface. 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. Interoperability of Galileo with the wide range of desktop software tools is assured by our use of highly interoperable components.

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 comprehend, 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 textual language. The benefit of this approach is easier and faster editing. 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.

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" win-

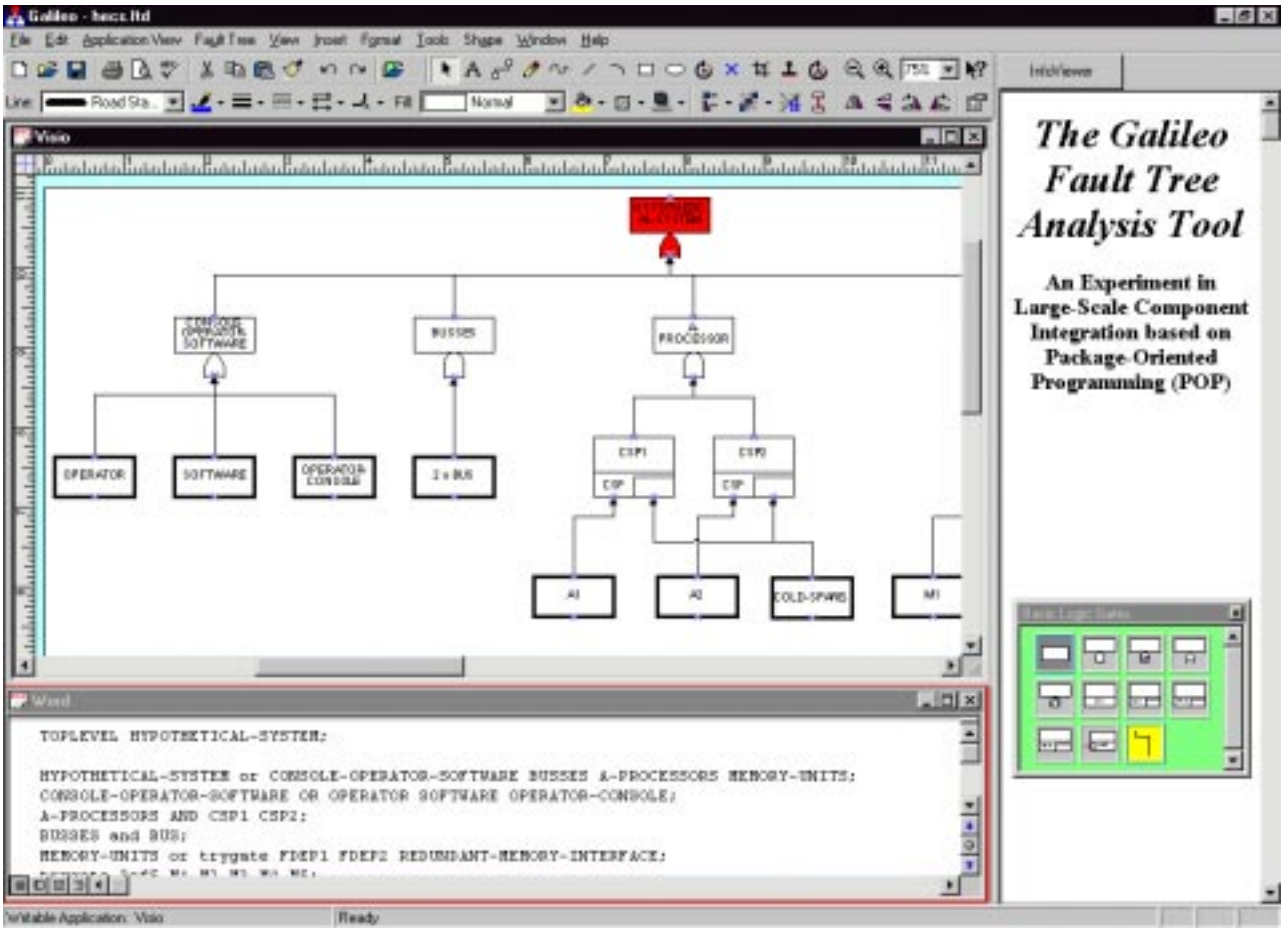

Figure 2. A screenshot of Galileo

dow. 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.

4. COMPONENT DESIGN WITH PACKAGES

A key feature of the software design of Galileo is the use of mass-market packages as components. We use the term Package-Oriented Programming (POP) [22][23][24] to describe the approach in which multiple packages are tightly integrated within a larger system. By using packages as components, POP leverages the vast investments that have been made in their design, construction, and evolution, and the tremendous economies of scale obtained by volume pricing. The approach can achieve characteristics at low cost that are otherwise prohibitively expensive: usability, rich function, familiarity and thus ease of learning, interoperability, documentation and support, and reasonably stable execution of these packages.

POP appears to be especially appropriate as an approach to building modeling and analysis tools. Tools consist of algorithmic analysis cores implemented in perhaps a few tens of thousands of lines of code. However, sophisticated 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 to dramatically lower the cost to develop and to use a tool which achieving high quality.

By using the approach to build Galileo we avoided designing a tremendous amount of code from scratch. Instead, we 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 about 30,000 lines of code, a reduction of several orders of magnitude (in both size and cost) compared to a build-from-scratch style producing a comparably useful result.

On the other hand, component-based development remains as much an aspiration in software engineering as a reality. Galileo serves as a concrete system with which we explore important issues in this area. Achieving such benefits through component-based design of complex software in a 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 goal for decades. Yet, with few exceptions, suc-

cess 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.

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 problems, 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.

5. 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 repre-

Table 1: Summary of subtree characteristics and solution methods

Parameters	Has Constant Probability?	*don't care*	Yes	No	No	No
	Static or Dynamic Tree?	Static	Dynamic	Dynamic	Dynamic	Dynamic
	Uses a Weibull Distribution	*don't care*	*don't care*	No	Yes	Yes
	Has Cold/ Warm Spare Gates?	N/A	*don't care*	*don't care*	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

sentation 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 then 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].

6. 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.

6.1 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

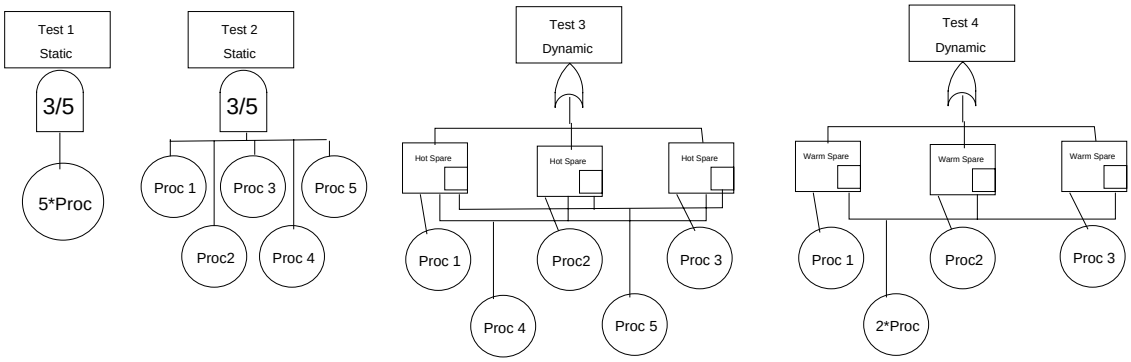

Figure 3. The four fault trees in this figure all produce the same numerical result

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 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.

6.2 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.

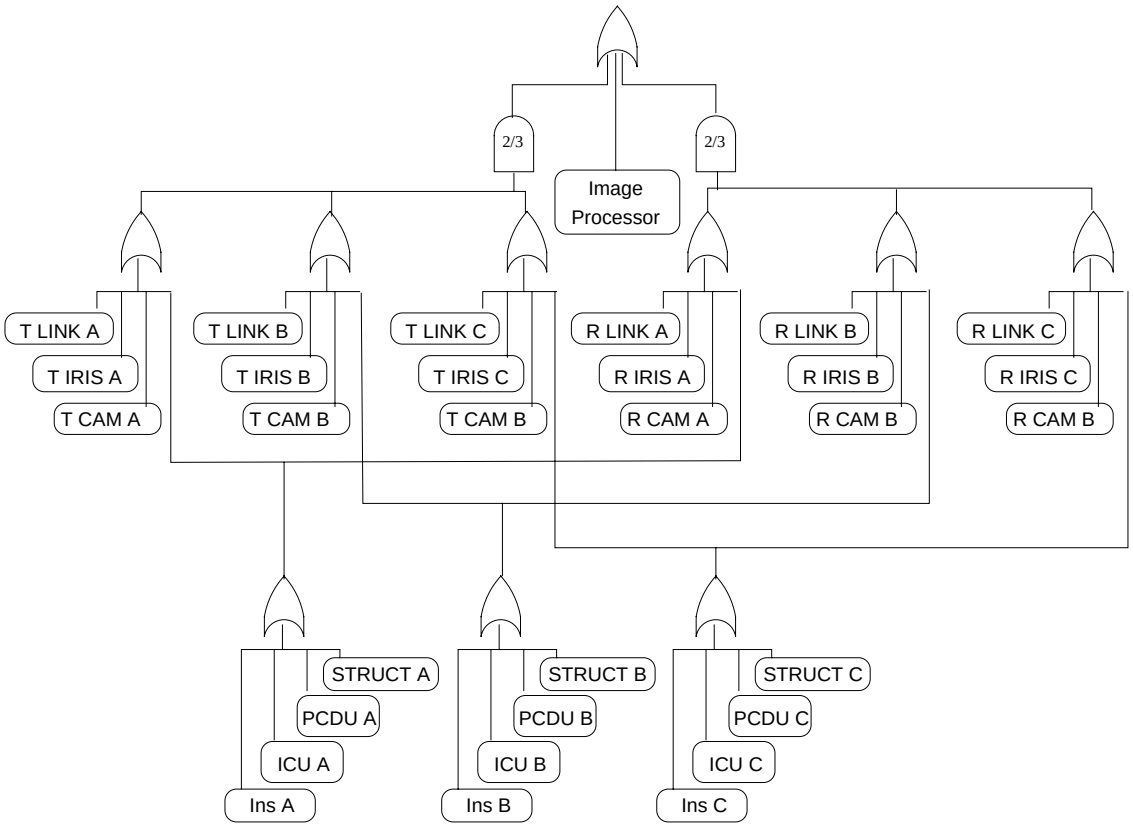

Figure 4. An example fault tree for testing

We believe that our approach to testing using diverse solution methods will help an analyst gain confidence in the results from our tool. We intend to expose our testing method through the tool interface, allowing the user to solve fault trees using all available methods (as opposed to the most efficient one) in order to compare results.

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).

7. FUTURE WORK

Our future efforts will continue in the two dimensions, software engineering and reliability engineering. The Galileo tool will continue to provide a common focus for our interdisciplinary research. In particular, on the basis of the work that we have done to date, we have agreed to develop an enhanced version of Galileo with NASA Langley Research Center. The new version will provide two main enhancements in the reliability engineering dimension. The first is support for the analysis of the reliability of systems that operate in phased missions. The second is sensitivity analysis of reliability as a function of variations in basic event probabilities.

In the tool and software engineering area, the advances will be both functional and non-functional. First, we will implement the new analysis features, of course. Second, we are developing a scalable approach to graphical layout and editing based on the commercial off-the-shelf Visio drawing component. We have already develop advanced prototypes of a new layout capability. Third, we are designing an enhanced domain-specific fault tree programming language based on principles of programming language design, emphasizing abstraction, modularity and composition. These aspects are critical to the scalable programming-level representation of fault trees for large systems. The new language will also support the representation of complex layouts for large fault trees.

In the non-functional software engineering area we are investing considerable effort in designing fault tree and fault tree analysis abstractions and implementations to help ensure demonstrable levels of software quality. We are especially concerned with the verifiability of the fault tree implementation, and with the evolvability of the resulting system.

Our approach to these issues is of course multi-faceted, but a key element is the use of Sullivan's mediator design concepts [27][28]. The technique provides a clean decomposition of a system into independent abstractions for basic representational elements, such as *fault tree*, *Markov chain*, *BDD* and *Graphical View*, and separate relational abstractions—the mediators—that serve to integrate them. These abstractions are

represented as objects separate from the objects that they relate. The system appears as an unrooted tree of objects, each implementing an independent representation, in a network of relational abstractions. For example, one mediator is responsible for creating and maintaining the correspondence between a *fault tree* object and a corresponding *Markov chain* object.

This approach provides important benefits. First, it permits us to reason about and to develop our system in a highly modular fashion. For example, we can develop and verify an object-oriented *Markov chain* class independently of our *fault tree* class, even though instances of these two classes are very tightly integrated at runtime. Second, it provides a flexibile software architecture that permits us to reconfigure our system quickly. For example, because the code that relates *fault tree* objects to *Markov chain* objects is external to those objects, we can replace the translation algorithm in a completely modular fashion. We are experimenting, in particular, with replacing the mediator that performs fault tree modularization.

In the non-fault-tree, software engineering area, we are focusing on abstracting a generic tool architecture for engineering modeling and analysis from the fault-tree-domain-specific Galileo tool. We seek to produce a generic framework for tools having the general features of Galileo, including the use of well known commercial packages as components, and support for integrated graphical drawing and domain-specific programming language features.

Galileo continues to function as a platform for delivering innovative reliability analysis techniques to engineers. We have recently agreed to produce a version of the tool for use in industrial settings with NASA Langley Research Center. Evaluating both our software development and reliability engineering techniques in the contexts of practice that this collaboration will enable promises to help us to deepen our understanding of fundamental issues in both research areas.

ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grants CCR-9502029 (CAREER), CCR-9506779, CCR-9804078, MIP 95-28258 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," to appear, *IEEE Trans. on Reliability*, 1999.

[2] Anju Anand and Arun K. Somani, "Hierarchical analysis of fault trees with dependencies, using decomposition," In *Proc. of the 1998 Reliability and Maintainability Symp.*, Jan 1998, pp 69-75.

[3] Barry W. Boehm and William L. Scherlis, "Megaprogramming," In *Proc. of the DARPA Software Technology Conference*, pp 63-82. Meridien Corp., Arlington, VA, 1992.

[4] Mark Boyd & Salvatore J. Bavuso, "Simulation modeling for long duration spacecraft control systems," *Proc. of the 1993 Reliability and Maintainability Symp.*, Jan 1993, pp 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, pp 101-137.

[7] David Coppit and Kevin J. Sullivan, "Formal specification in collaborative design of critical software tools," In *Proc. Third IEEE Int'l High-Assurance Systems Engineering Symp.*, pp 13-20, Washington, D.C., 13-14 Nov 1998. IEEE.

[8] O. Coudert and J.C. Madre, "Fault tree analysis: 10^{20} prime implicants and beyond," In *Proc. of the Reliability and Maintainability Symp.*, 1993, pp 240-245.

[9] Stacy A. Doyle and Joanne Bechta Dugan, "Dependability assessment using binary decision diagrams," In *Proc. of the IEEE Int'l Symp. 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*, vol 39, pp 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*, vol 41, num 3, pp 363-377, Sep 1992.

[12] Joanne Bechta Dugan, Bharath Venkataraman and Rohit Gulati, "DIFTree: A software package for the analysis of dynamic fault tree models," *Proc. of the 1997 Reliability and Maintainability Symp.*, Jan 1997, pp 64-70.

[13] Yves Dutuit and Antoine Rauzy, "A linear-time algorithm to find modules in fault trees," *IEEE Transactions on Reliability*, Sep 1996.

[14] David Garlan, Robert Allen, and John Ockerbloom. "Architectural mismatch: Why reuse is so hard." *IEEE Software*, vol 12, num 6, pp 17-26, Nov 1995.

[15] Rohit Gulati and Joanne Bechta Dugan, "A modular approach for analyzing static and dynamic fault trees," in *Proc. of the Reliability and Maintainability Symp.*, Jan 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, Int'l 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," *Proc. of the 1999 Reliability and Maintainability Symp.*, Jan 1999, pp 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*, vol 43, June 1980, pp 136-138.

[22] K.J. Sullivan and J.C. Knight, "Building Programs from Massive Components," in *Proc. 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," *Proc. of the 18th Int'l Conference on Software Engineering*, Berlin, Mar 1996, pp 220-229.

[24] Kevin J. Sullivan, Jake Cockrell, Shengtong Zhang, and David Coppit, "Package-oriented programming of engineering tools," In *Proc. of the 19th Int'l Conference on Software Engineering*, pp 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," Proc. of the 29th Int'l 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] K.J. Sullivan and D. Notkin, "Reconciling Environment Integration and Software Evolution," *ACM Transactions on Software Engineering and Methodology 1*,3, July 1992, pp. 229—268.

[28] K.J. Sullivan, I.J. Kalet and D. Notkin, "Evaluating the Mediator Method: Prism as a Case Study," *IEEE Transactions on Software Engineering 22*,8, August 1996, pp. 563—579.

[29] United States Nuclear Regulatory Commission, *Fault Tree Handbook*, NUREG-0492, 1981.

[30] H.A. Watson and Bell Telephone Laboratories, "Launch Control Safety Study," Bell Telephone Laboratories, Murray Hill, NJ USA, 1961.

AUTHORS

Joanne Bechta Dugan; Department of Electrical Engineering; Thornton Hall, University of Virginia; Charlottesville, VA 22903 USA.
Internet (e-mail): jbd@ virginia.edu

Joanne Bechta Dugan was awarded the B.A. degree in Mathematics and Computer Science from La Salle University, Philadelphia, PA in 1980, and the M.S. and Ph.D degrees in Electrical Engineering from Duke University, Durham, NC in 1982 and 1984, respectively. She has served on the faculty since 1993. She is an associate editor of the IEEE Transactions on Reliability and a member of the National Research Council's committee on application of digital instrumentation and control systems to nuclear power plant operations and safety. She is also a member of IEEE, Eta Kappa Nu, and Phi Beta Kappa. Previously, she taught at Duke University and worked as a visiting scientist at the Research Triangle Institute.

Kevin J. Sullivan; Department of Computer Science; Thornton Hall, University of Virginia; Charlottesville, VA 22903 USA
Internet (e-mail): sullivan@cs.virginia.edu

Kevin J. Sullivan received the dual BA degree in Mathematics and in Computer Science from Tufts University in 1987, the MS in Computer Science and Ph.D. in Computer Science and Engineering from the University of Washington in 1990 and 1994, respectively. Since then he has been Assistant Professor in the Department of Computer Science at the University of Virginia. His primary research area is software engineering, with a focus on software design. He studies modularity and evolution, in particular. He currently has projects in component-based design, software engineering economics, infrastructure survivability, and software evolution.

David Coppit; Department of Computer Science; Thornton Hall, University of Virginia; Charlottesville, VA 22903 USA
Internet (e-mail): david@coppit.org

David Coppit is a Ph. D. student at the University of Virginia. He received a Bachelor of Science degree in Computer Science and another in Physics at the University of Mississippi, Oxford, Mississippi, in 1995. His field of study is software engineering, with a particular interest in software development using large-scale components.