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

Joanne Bechta Dugan, *Fellow, IEEE*, Kevin J. Sullivan, *Member, IEEE*, and David Coppit

***Abstract*—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 widespread use. A critical impediment to disseminating new methods is the inability to produce, at a reasonable cost, supporting software tools that have the**

- **usability and dependability characteristics that industrial users require,**
- **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, and thus costly programs. Tools on the order of a million lines of code are typical, with much of the code devoted to

- **tool interoperability,**
- **human-computer interface,**
- **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,**
- **essentially unmaintainable.**

This paper presents 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 Galileo: a fault tree modeling and analysis tool. These innovations are described in 2 dimensions

1) **The Galileo core reliability modeling and analysis function.**
2) **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 seldom applies 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,**
- **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[1] sub-trees;**
- **translates them into appropriate submodels based on Markov chains, Boolean decision diagrams, and other formalisms,**
- **analyzes the submodels;**
- **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.

This software engineering approach centers on the component-based design techniques of Sullivan and his colleagues. A key element of the approach is the use of mass-market software packages as large components, *viz*, package-oriented programming. It achieves at low cost

- **an effective human-computer interface,**
- **tool interoperability,**
- **considerable dependability for the function delivered.**

Low-cost means that the effort involves a small handful of graduate and undergraduate students and faculty. Sullivan's mediator-based design approach is also used 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 validate this modeling approach and to verify its implementation, both natural-language and formal specifications are being developed for the fault-tree gates and their interactions.

Galileo has been evaluated 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, the redundancy inherent in the use of multiple analysis techniques in Galileo is used as an aid to testing this software. Galileo has been acquired by hundreds of sites. We are now building an enhanced version with NASA Langley Research Center.

***Index Terms*—Component software, Fault-tree analysis, Reliability analysis, Software engineering, Testing.**

I. INTRODUCTION

Acronyms[2]

BDD	binary decision diagram
POP	package-oriented programming
FDEP	functional dependency gate

Manuscript received August 31, 1999; revised October 5, 1999. This work was supported in part by the US 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.

The authors are with the Department of Electrical Engineering, Thornton Hall, University of Virginia, Charlottesville, VA 22903 USA (e-mail: JBD@virginia.edu; Sullivan@cs.virginia.edu; David@Coppit.org).

Publisher Item Identifier S 0018-9529(00)06202-3.

[1]"Independent" is used in its nonstatistical meaning in this paper.

[2]The singular and plural of an acronym are always spelled the same.

PDEP	probabilistic dependency gate
PAND	priority-and gate
SEQ	sequence enforcing gate
FTA	fault-tree analysis
FTCS	fault-tolerant computer system
MS	Microsoft

Analysts concerned with quantitative assessment of reliability or safety of FTCS have a variety of mathematical techniques at their disposal: e.g., 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, seldom does a single technique apply to an entire system, because of the

- size of the system,
- 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,
- 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. These tools must

- support rich functions such as graphical editing, persistent storage, report generation, and data management;
- have a level of usability best acquired through "usability engineering" and "conformance to standard user-interface conventions;"
- run on industrial computing platforms of choice—today Windows-based workstations;
- be interoperable, i.e., integrate cleanly with other software systems typically used on those platforms and within the broader engineering activities of an engineering enterprise;
- be developed at low cost to avoid prohibitive pricing [26], because the markets for such tools are small.

In addition to providing analytic 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 assurance that models faithfully represent systems,
- to provide a sound basis for software design and implementation.

The validity of models and their analytic results is especially important if a tool is used to develop mission-critical or safety-critical applications [7].

This paper presents our approach to developing Galileo, a high-quality tool for dynamic FTA, which addresses these concerns.

1) Our analysis methodology, `DIFTree` [11], [15], uses a modular approach that automatically combines several different analysis techniques.
2) We use 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 Galileo uses a component-based software development approach that we call "POP." In this style, a few large-scale, widely used, volume priced software packages provide the vast bulk of the nonanalysis functions at low cost while meeting the critical functional, usability, and interoperability requirements for sophisticated tools [23]–[25]. We are developing a combination of natural-language[3] and partial formal specifications for the fault-tree gates and their interactions to—

- help validate the modeling framework,
- aid users in building valid models,
- provide a basis for verifying the implementation of the analysis approach.

We exploit the inherent redundancy associated with multiple analysis techniques as an aid in testing.

Section II provides background on dynamic FTA and the software engineering problems involved in producing tools and ultimately in making advances in engineering modeling and analysis.

Section III presents the concrete results of our work, the Galileo fault tree tool.

Section IV discusses a central feature of our software engineering approach, namely the use of mass-market software packages as components.

Section V discusses our use of natural-language and formal specifications to help validate our fault-tree modeling framework and verify the Galileo tool implementation.

Section VI characterizes the use of various FTA techniques for solution and for testing.

Section VII outlines future work.

II. Background

A. Dynamic FTA

Fault trees [16], [30] were developed to facilitate unreliability analysis of the Minuteman missile system [31]. 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 method of choice for reliability analysis of fault tolerant systems after

[3]A "natural-language" is an ordinary hereditary language spoken by a group of individuals as their native tongue. Contrast this with "artifical languages" like programming languages.

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 FTA. The complex redundancy-management techniques typically used in FTCS (e.g., 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 FTCS is thus frequently based on Markov models. Fault trees remain a popular modeling choice for reliability analysis of nonfault tolerant systems. Most reliability engineers are well-versed in FTA.

Recent work in dynamic fault trees has addressed both of these limitations and has resulted in an FTA approach that applies to FTCS and nonfault 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,
- 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 BDD 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 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 are also 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 [13] for finding independent subtrees (subtrees which share no basic events). The algorithm identifies independent subtrees during a depth-first traversal of the tree, and records 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 the Rauzy algorithm [13] on the fault tree model, one can

- automatically detect independent subtrees,
- classify them as static or dynamic,
- 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 3 separate Markov models each of 10^3 states with the solution of the combined model, containing a cross-product of each state space, and thus 10^9 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 FTA methodology [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. `DIFTree` fully supports coverage modeling in static and dynamic subtrees. Failure probabilities in static subtrees can be constant (time-independent) or follow the exponential distribution. Dynamic trees support only the exponential distribution of time to failure.

Fig. 1 illustrates the modularization operation of `DIFTree` on a hypothetical fault tree containing

- 2 static subtrees, and
- 2 dynamic subtrees.

The static subtrees are solved by automatic conversion to the equivalent BDD; 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 4 basic events, one representing each subtree. This example is 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, an exact solution can not be provided for a subdivided dynamic subtree. Thus, to avoid the use of an unbounded approximation, `DIFTree` does not split dynamic subtrees. This is a choice of accuracy over performance-time, as the further splitting of a dynamic subtree can substantially improve solution time. Ref. [2] presents a similar technique which does split dynamic subtrees, trading accuracy for performance.

B. *Software Engineering of Modeling Tools*

Algorithmic advances in engineering modeling and analysis have little chance of being validated effectively or having an important 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 at least 10^6 lines of code, and subject to demanding functional, usability, interoperability, and dependability requirements. Specifying, designing, implementing, verifying, correcting, and enhancing them requires software engineering expertise and large, often prohibitive, investments.

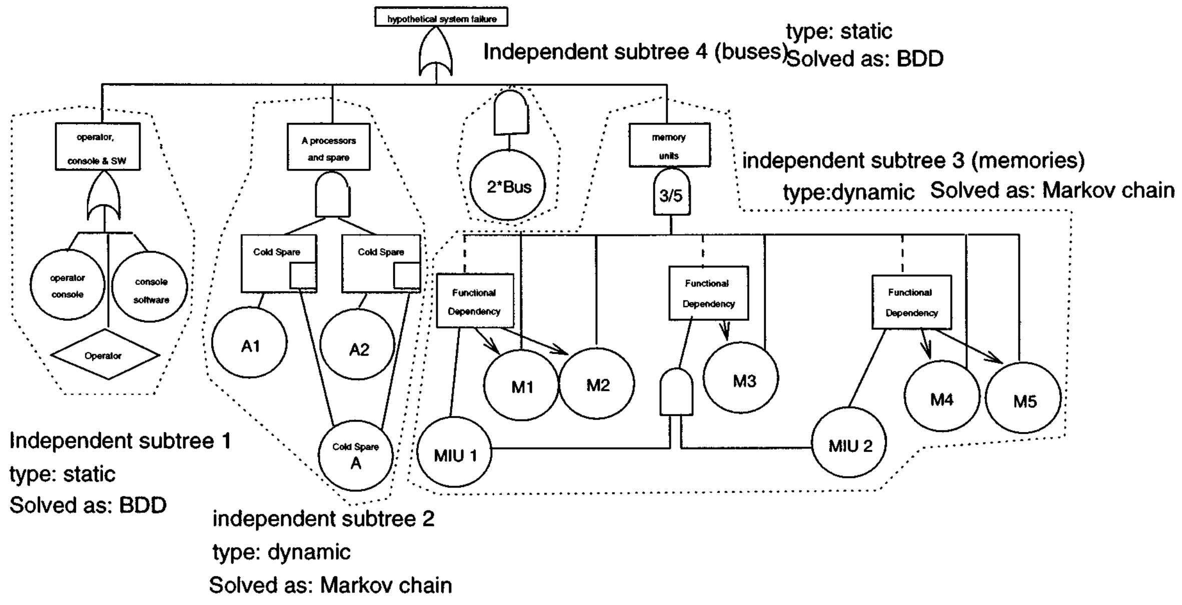


Fig. 1. An Example Modularization

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 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 is rapidly becoming apparent.

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 appreciable use in practice.

The problem goes deeper. 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 recognizing 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, POP, provides the vast bulk of nonanalysis software using low-cost, richly functional, commercial components, thus reducing by orders of magnitude the software that must be built from scratch. The research problem is to understand and overcome important impediments to component-based software design.

Even if successful, using packages is not a panacea. Several major challenges remain

- the specification and validation of modeling and analysis frameworks—see Section III;
- devising 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 the Sullivan mediator design approach [28], [29] extensively in structuring this code for modular development and ease of evolution.

III. The User View of Galileo

Fig. 2 presents the user-view of Galileo, our primary software artifact [26]. In the upper left is a graphic representation of the fault tree. The same tree is expressed in a domain-specific-language-based text form in the lower left. The window on the right displays documentation and can be used to contact the tool authors. Each of the views is integrated into the main Galileo window.

The Galileo tool architecture uses

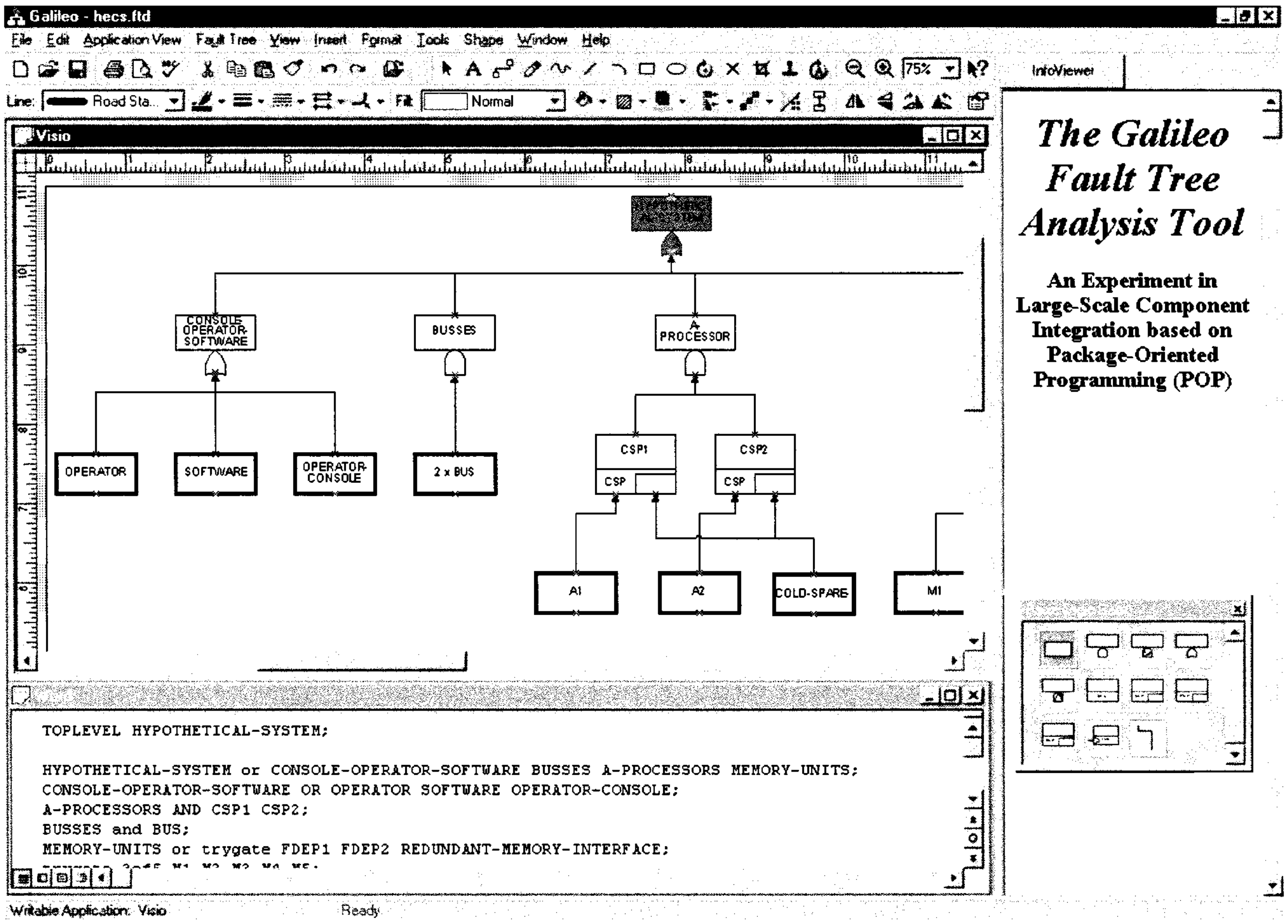
- Visio Corporation's Visio Technical,[4]
- MS Word,
- MS Internet Explorer,

to create much of the human-computer interface. Users benefit from the tremendous investment in the design and implementation of these components. For example,

- MS Word supports find-and-replace;
- Visio Technical supports panning, zooming, and cut-and-paste of graphical views, etc.,
- Visio Technical 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 the use of highly interoperable components.

[4]Visio Corporation has recently been acquired by Microsoft Corp.


Galileo - hecs.ftd
File Edit Application View Fault Tree View Insert Format Tools Shape Window Help
InfoViewer
Visio
CONSOLE OPERATOR SOFTWARE
BUSSES
A PROCESSOR
OPERATOR
SOFTWARE
OPERATOR CONSOLE
2 x BUS
CSP1
CSP2
CSP
A1
A2
COLD-SPARE
M1
Writable Application: Visio
Ready

The Galileo Fault Tree Analysis Tool

An Experiment in Large-Scale Component Integration based on Package-Oriented Programming (POP)

```
TOPLEVEL HYPOTHETICAL-SYSTEM;

HYPOTHETICAL-SYSTEM or CONSOLE-OPERATOR-SOFTWARE BUSSES A-PROCESSORS MEMORY-UNITS;
CONSOLE-OPERATOR-SOFTWARE OR OPERATOR SOFTWARE OPERATOR-CONSOLE;
A-PROCESSORS AND CSP1 CSP2;
BUSSES and BUS;
MEMORY-UNITS or trygate FDEP1 FDEP2 REDUNDANT-MEMORY-INTERFACE;
```

Fig. 2. A Screen-Shot of Galileo

Galileo supports two views of fault trees.

1) 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 graphic nature makes it easy to comprehend, although it is not as easy to edit for some people as the textual view.
2) The textual view describes the same fault tree using a textual language. This approach is easier and faster to edit; but is more difficult to comprehend.

Each of the packages we used was specialized for use in our POP-based architecture. For example, we customized Visio Technical by

- creating a *stencil* of fault-tree shapes and connectors,
- 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 the MS Active Document approach to containing multiple documents (a Visio Technical drawing, an MS Word document, and an Explorer browser) in an overall *container* window. The container ensures that

- the menus of the currently active document are displayed,
- Galileo-specific menu choices are merged into the menus of the packages.

The Galileo menus allow the user automatically to propagate 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.

IV. Component Design With Packages

A key feature of the software design of Galileo is the use of mass-market packages as components. We use POP [23]–[25] 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,
- support,
- 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 10^4 of lines of code. However, sophisticated tools also have a superstructure supporting such features as textual and graphical interfaces and report generation; these can not be implemented 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 use a tool which achieves high quality.

By using the POP approach to build Galileo, we avoided designing a tremendous amount of code from scratch. Instead, we designed and implemented

- a fault-tree data type,
- 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 built about 30 k 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, 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 provide only 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. A widely cited paper [14] 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. The authors of [14] 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 [17] argued that the component dream was unlikely to be realized for 3 reasons

- components make conflicting assumptions;
- components are costly to develop;
- components are costly to understand.

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.

POP is an attempt to thread the eye of this needle. We agree that success requires the use of large components: only they can 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: *viz*, 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;
- 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 research on POP.

At this point, the question arises: Do the difficulties that we encountered rise to the level of the problems of architectural mismatch that was observed [14] 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 be, and can only be, enabled in general by conformance to shared design rules, or integration architectures [24], [27]. We undertook the work reported here knowing that the components that we are using all conform to a common integration architecture: the MS 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 MS Word and Visio Technical, 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

- the virtual machines presented by the packages were not always what we needed to implement the functions that we wanted;
- in some cases, the packages did not fully conform 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 difficulties we have experienced, the approach appears to have considerable potential to enable important advances in the production of complex systems in some important domains, especially that of engineering modeling and analysis tools. The approach—

- addresses the Lampson concern for development cost by using volume-priced packages as components;
- 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

- the specifications of the exposed virtual machines are not well documented;
- the components evolve continually as versions are released.

Of course, such evolution is double-edged: it presents difficulties, but also presents the opportunity to give major new capabilities to end users at extremely low cost.

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.

V. 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 (FDEP, PAND, SEQ, and Cold Spare) has been expanded to include three more (Hot Spare, Warm Spare, and PDEP). FDEP and PDEP are used to model (deterministic and probabilistic, respectively) cascading (or common-cause) failures. PAND and SEQ 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 FTA can be applied, because the sequential behavior characteristic of FTCS can be effectively captured in a dynamic fault tree. Dynamic fault trees are solved by automatic conversion to equivalent Markov models [10]. However, using dynamic fault trees in an industrial setting raised two concerns: How can the analyst

1) know that a model is an accurate representation of the system being analyzed?
2) be assured 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,
- verifying an implementation,
- producing user documentation,

we are developing partial formal specifications of dynamic fault trees in the Z[5] language. An early version contains [7]

- formal specifications of static and dynamic gates,
- how each is evaluated at a given system state,
- the permitted structure of a dynamic fault tree as a composition of basic events and gates.

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. Forthcoming work will provide a far more comprehensive formal semantics for dynamic fault-trees.

However, the formal specification of gates does not necessarily help a reliability engineer gain assurance 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 are also deriving, from the formal specification, a set of carefully worded natural-language specifications for each gate. 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].

VI. Combining Analysis Techniques

Our approach to the solution of fault-trees automatically decomposes the system-level fault-tree into modules that are solved separately. This modular approach allows different subtrees to be solved by different methods [15]

- static subtrees can be solved by conversion to an equivalent BDD,
- dynamic subtrees can be solved by conversion to the equivalent Markov chain.

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 offers interesting possibilities in terms of multiple solutions of the same subtree.

This section discusses

- the decision criteria for choosing a particular solution algorithm,
- how we exploited the alternatives as an aid in testing.

A. *Choosing Appropriate Analysis Techniques*

Table I summarizes the applicability of several dynamic FTA techniques. The upper half of Table I shows combinations of 4 characteristics of subtrees: whether—

- a subtree uses constant failure probabilities (as opposed to a distribution of time to failure),
- it has any dynamic gates,
- it has any cold or warm spare gates,
- it uses a Weibull time-to-failure distribution.

The lower half of Table I describes the abilities of the solution alternatives, given the 4 characteristics above it.

Traditional cut-set approaches to FTA apply only to static fault-trees and are generally inferior to the newer BDD based approaches.

[5]Z (pronounced "Zed") is a formal specification notation based on typed set theory and on first-order predicate logic [22].

TABLE I
SUMMARY OF SUBTREE CHARACTERISTICS AND SOLUTION METHODS

Parameters					
Has Constant Probability	*don't care*	Yes	No	No	No
Tree Type	Static	Dynamic	Dynamic	Dynamic	Dynamic
Uses a Weibull Distribution	*don't care*	*don't care*	No	Yes	Yes
Has Cold/Warm Spare Gates	not applicable	*don't care*	*don't care*	No	Yes
Analytic Techniques					
Cut Sets	Possible	NOT ALLOWED	NOT POSSIBLE	NOT POSSIBLE	NOT POSSIBLE
BDD	Preferred	NOT ALLOWED	NOT POSSIBLE	NOT POSSIBLE	NOT POSSIBLE
Markov Chains	Possible**	NOT ALLOWED	Preferred	Possible	Not Feasible
Monte Carlo Simulation	Possible	NOT ALLOWED	Possible	Possible	Preferred

** If no constant probabilities.

Markov methods apply 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.

Monte Carlo simulation presents a viable alternative to the analytic approaches in several interesting situations. The combination of warm or cold spares and Weibull (or other nonexponential) 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-out-of-12 gate, where each of the 12 inputs was a 5-out-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 Monte Carlo 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 (constant probability of failure in a dynamic model) are disallowed. If simulation is added to the set of solvers, some previously disallowed situations (cold or warm spares, and Weibull time-to-failure) are now permissible. Monte-Carlo simulation applies to both static and dynamic subtrees and in some cases (e.g., large combinatorics) can be more attractive than the analytic approach.

B. Solution Techniques As An Aid In Testing

Because multiple solution techniques are available within a single tool, we can exploit this flexibility in creating test cases, in two different ways.

1) For some trees, multiple solution techniques apply, although one can 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; the Markov approach is based on differential equations), there is a basis for greater assurance that the results are correct if both solutions produce the same results. Further, some of these structures 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. Thus we created a set of test cases that exploit these similarities.
2) 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 PAND.

As an example of the use of the diverse solution methods for testing, consider the 4 fault trees in Fig. 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.
- Test case 4 uses the warm spare gate with a dormancy factor of 1.[6]

[6]The dormancy factor ($\in [0, 1]$) represents the reduction in the failure rate experienced while the spare is dormant; 0 corresponds to a cold spare, 1 to a hot spare.

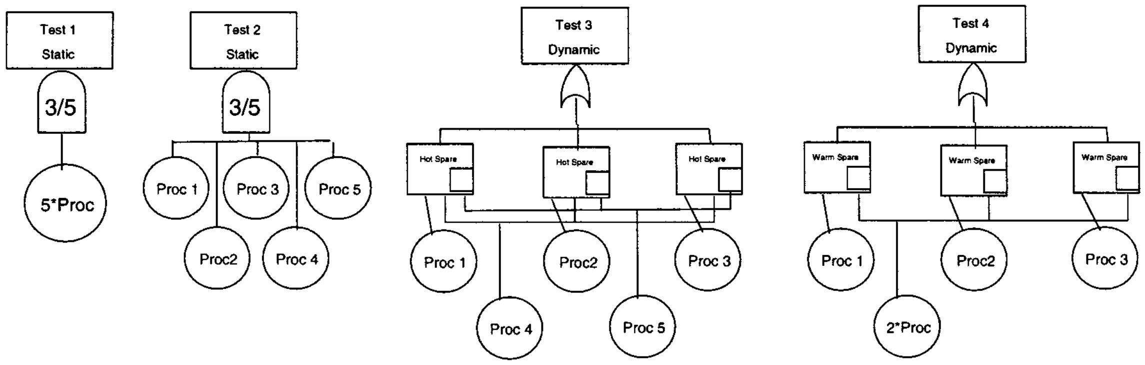


Fig. 3. These 4 Fault Trees All Produce the Same Numerical Result

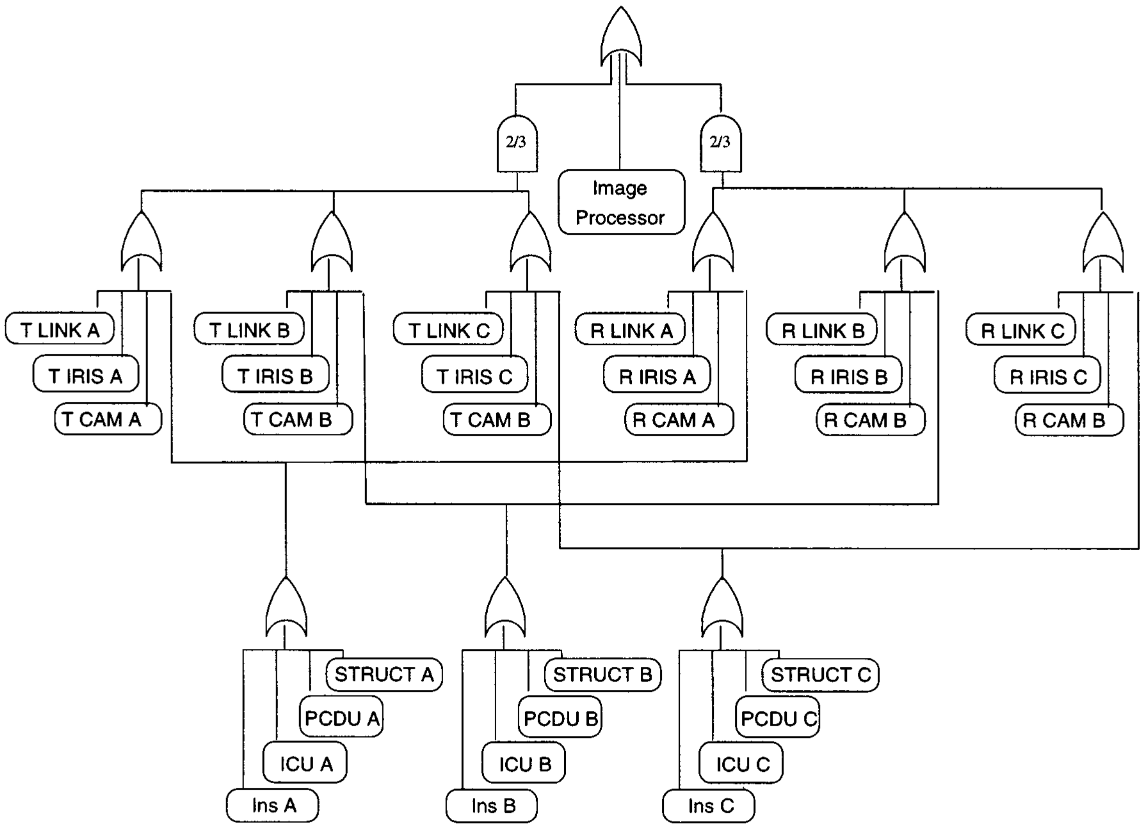


Fig. 4. Example Fault-Tree for Testing

Test cases 1 and 2 are static models; test cases 3 and 4 are dynamic models. The test set can be extended by varying the

- coverage parameters (perfect vs. imperfect),
- failure distribution (the degenerate case of the Weibull distribution is the exponential).

We believe that our approach to testing using diverse solution methods will help an analyst develop assurance 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. Fig. 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 FTA packages do not necessarily produce correct results. In fact, both of the commercial packages produced the same incorrect result because they could not recognize internal (nonbasic) events which fan out (are used as input to more than one gate).

VII. Future Work

Our future effort will continue in the 2 dimensions,

- software engineering,
- reliability engineering.

The Galileo tool continues to provide a common focus for our interdisciplinary research. In particular, on the basis of our work 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

- Support for the analysis of the reliability of systems that operate in phased missions.
- 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 nonfunctional

- We will implement the new analysis features, of course.
- We are developing a scalable approach to graphical layout and editing based on the commercial off-the-shelf Visio Technical drawing component. We have already developed advanced prototypes of a new layout capability.
- 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. This new language will also support the representation of complex layouts for large fault trees.

In the nonfunctional software engineering area we are investing considerable effort in designing fault tree and FTA abstractions and implementations to help ensure demonstrable levels of software quality. We are especially concerned with the

- validity of the fault-tree modeling framework,
- verifiability of the fault-tree implementation,
- evolvability of the resulting system.

Our approach to these issues is multi-faceted. We are developing a comprehensive formal semantics of dynamic fault-tree models for framework validation and program implementation. A key element in evolvability is the use of the Sullivan mediator design concepts [28], [29]. 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

- 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.
- It provides a flexible 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 to support the use of various modularization policies.

In the software engineering area we are also 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,
- 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 deepen our understanding of fundamental issues in both research areas.

Acknowledgment

The authors would like to thank the reliability engineers at NASA Langley Research Center and Lockheed-Martin Corporation for their valuable feedback.

References

[1] S. Amari, J. B. Dugan, and R. Misra, "A separable method for incorporating imperfect coverage into combinatorial models," *IEEE Trans. Reliability*, vol. 48, pp. 267–274, Sept. 1999.
[2] A. Anand and A. K. Somani, "Hierarchical analysis of fault trees with dependencies, using decomposition," *Proc. Ann. Reliability and Maintainability Symp.*, pp. 69–75, 1998.
[3] B. W. Boehm and W. L. Scherlis, "Megaprogramming," *Proc. DARPA Software Technology Conf*, pp. 63–82, 1992.
[4] M. Boyd and S. J. Bavuso, "Simulation modeling for long duration spacecraft control systems," in *Proc. Ann. Reliability and Maintainability Symp.*, 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," in *Reliability and Fault Tree Analysis*: SIAM, 1975, pp. 101–137.
[7] D. Coppit and K. J. Sullivan, "Formal specification in collaborative design of critical software tools," in *Proc. Third IEEE Int'l. High-Assurance Systems Engineering Symp.*, Washington, DC, Nov. 1998, pp. 13–20.
[8] O. Coudert and J. C. Madre, "Fault tree analysis: 10^{20} prime implicants and beyond," in *Proc. Ann. Reliability and Maintainability Symp.*, 1993, pp. 240–245.
[9] S. A. Doyle and J. B. Dugan, "Dependability assessment using binary decision diagrams," in *Proc. IEEE Int'l Symp. Fault-Tolerant Computing*, vol. FTCS-25, June 1995.
[10] J. B. Dugan, S. Bavuso, and M. 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] ——, "Dynamic fault tree models for fault tolerant computer systems," *IEEE Trans. Reliability*, vol. 41, pp. 363–377, Sept. 1992.
[12] J. B. Dugan, B. Venkataraman, and R. Gulati, "DIFTree: A software package for the analysis of dynamic fault tree models," in *Proc. Ann. Reliability and Maintainability Symp.*, 1997, pp. 64–70.
[13] Y. Dutuit and A. Rauzy, "A linear-time algorithm to find modules in fault trees," *IEEE Trans. Reliability*, vol. 45, pp. 422–425, Sept. 1996.
[14] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: Why reuse is so hard," *IEEE Software*, vol. 12, pp. 17–26, Nov. 1995.
[15] R. Gulati and J. B. Dugan, "A modular approach for analyzing static and dynamic fault trees," in *Proc. Ann. Reliability and Maintainability Symp.*, 1997, pp. 57–63.

[16] E. J. Henley and H. Kumamoto, *Probabilistic Risk Assessment*: IEEE Press, 1992.
[17] B. Lampson, "How software components grew up and conquered the world," in *Keynote address, Int'l Conf. Software Engineering*, May 1999.
[18] R. Manian, D. Coppit, K. J. Sullivan, and J. B. Dugan, "Bridging the gap between systems and dynamic fault tree models," in *Proc. Ann. Reliability and Maintainability Symp.*, 1999, pp. 105–111.
[19] Microsoft, "Active Document Containers,", http://msdn.microsoft.com/library/devprods/vs6/visualc/ vccore/_core_activex_document_containers.htm.
[20] D. Rogerson, *Inside COM*: Microsoft Press, 1996.
[21] A. Rosenthal, "Decomposition methods for fault tree analysis," *IEEE Trans. Reliability*, vol. 43, pp. 136–138, June 1980.
[22] J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd ed: Prentice-Hall Int'l. Series in Computer Science, 1992.
[23] K. J. Sullivan and J. C. Knight, "Building programs from massive components," in *Proc. 21st Ann. Software Engineering Workshop*, Greenbelt, MD, Dec. 4–5, 1996.
[24] ——, "Experience assessing an architectural approach to large-scale, systematic reuse," in *Proc. 18th Int'l Conf. Software Engineering*, Berlin, Mar. 1996, pp. 220–229.
[25] K. J. Sullivan, J. Cockrell, S. Zhang, and D. Coppit, "Package-oriented programming of engineering tools," in *Proc. 19th Int'l Conf. Software Engineering*, Boston, May 1997, pp. 616–617.
[26] K. J. Sullivan, J. B. Dugan, and D. Coppit, "The Galileo fault tree analysis tool," in *Proc. 29th Int'l Conf. Fault-Tolerant Computing*, vol. FTCS-29, 1999.
[27] K. J. Sullivan, M. Marchukov, and J. Socha, "Analysis of a conflict between aggregation and interface negotiation in Microsoft's component object model," *IEEE Trans. Software Engineering*, pp. 584–599, July/Aug. 1999.
[28] K. J. Sullivan and D. Notkin, "Reconciling environment integration and software evolution," *ACM Trans. Software Engineering and Methodology*, pp. 229–268, July 1992.
[29] K. J. Sullivan, I. J. Kalet, and D. Notkin, "Evaluating the mediator method: Prism as a case study," *IEEE Trans. Software Engineering*, vol. 22, pp. 563–579, Aug. 1996.
[30] United States Nuclear Regulatory Commission, *Fault Tree Handbook*, 1981.
[31] H. A. Watson and Bell Telephone Labs, "Launch Control Safety Study," Bell Telephone Laboratories, 1961.

Joanne Bechta Dugan was awarded the B.A. (1980) in Mathematics and Computer Science from La Salle University, Philadelphia, and the MS (1982) and Ph.D. (1984) in electrical engineering from Duke University, Durham. She has served on the U.Va. faculty since 1993. She was 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 a Fellow of IEEE, a member of 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 (M) received the B.S. (1987) in Mathematics and in Computer Science from Tufts University, the M.S. (1990) in Computer Science, and Ph.D. (1994) in Computer Science and Engineering from the University of Washington. 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. He has projects in component-based design, software engineering economics, infrastructure survivability, and software evolution.

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

The author has requested enhancement of the downloaded file. All in-text references underlined in blue are linked to publications on ResearchGate.