

# Sound Methods and Effective Tools for Engineering Modeling and Analysis

David Coppit  
Department of Computer Science  
McGlothlin-Street Hall  
The College of William and Mary  
Williamsburg, VA 23185 USA  
+1 757 221 3476  
david@coppit.org

Kevin J. Sullivan  
Department of Computer Science  
151 Engineer's Way, P.O. Box 400470  
University of Virginia  
Charlottesville, VA 22904-4740 USA  
+1 434 982 2206  
sullivan@cs.virginia.edu

## Abstract

*Modeling and analysis is indispensable in engineering. To be safe and effective, a modeling method requires a language with a validated semantics; feature-rich, easy-to-use, dependable tools; and low engineering costs. Today we lack adequate means to develop such methods. We present a partial solution combining two techniques: formal methods for language design, and package-oriented programming for function and usability at low cost. We have evaluated the approach in an end-to-end experiment. We deployed an existing reliability method to NASA in a package-oriented tool and surveyed engineers to assess its usability. We formally specified, improved, and validated the language. To assess cost, we built a package-based tool for the new language. Our data show that the approach can enable cost-effective deployment of sound methods by effective tools.*

## 1. Introduction

Modeling and analysis methods are at the heart of engineering. Such a method is based on a modeling language for describing systems, with a semantics in a mapping of expressions (models) to estimates of system properties (results). Functions for manipulating and analyzing models are then generally supported by software tools.

For a method to be safe and effective, its semantics must be well defined and it must be supported by high-quality tools. Semantic soundness demands an abstract, precise, complete, and validated specification of the mapping of models to results. High quality tools, on the other hand, are easy to use, rich in function, and reliable. Such a tool must provide a broad set of easy-to-learn-and-use functions matched to the needs of *engineering practice*. These functions go well beyond prototype-style barebones editing and analysis, to include printing of large

models on engineering-sized paper, cut-and-paste into presentation tools, sophisticated graphical editing, etc. Engineers expect usability and functionality on par with mass-market packages, and demand it even in specialized tools. Reliability, by contrast, demands a sound semantic specification and a tool that implements it faithfully.

This paper emerges from a long-term, end-to-end experimental research program addressing the problem that, hampered by shortcomings in software engineering and languages, we are unable to deliver semantically sound methods supported by high quality tools at low cost. By experimental, we mean that we build and evaluate real tools using new techniques. By end-to-end, we mean that our work spans the life cycle and involves real customers, against whose needs results are evaluated.

Here, we address two sub-problems. First, we lack *demonstrably cost-effective* ways to ensure semantic soundness of methods too complex for informal specification. Second, we lack *low-cost* approaches to producing easy-to-use, functionally rich tools. Reliability will be addressed in forthcoming work. The contribution of this work is to show that we can, at low cost, combine well known formal methods [9] with package-oriented programming [30] to create semantically sound languages supported by richly functional, industrially usable tools. We report on the evaluation of a package-based tool, Galileo, by NASA engineers, and on the development of a similar tool, Nova, based on a formal semantics.

The rest of this paper is organized as follows. Section 2 discusses the role that formal specification should play in the development of a modeling method. Section 3 describes our approach and basic results. Section 4 introduces our experimental application domain. Section 5 describes the development of Nova using the combined approach. Section 6 presents results from the end-user evaluation of Galileo. Section 7 evaluates our work. Section 8 addresses related work. Section 9 concludes.

## 2. Formal Semantics and Dependability

A modeling method rests on a mapping of models to analysis results. Such a mapping should be defined in both specification and implementation forms. Of the two, the specification is the more fundamental. Lacking a specification, validation of the method is difficult, there is no basis for a definitive user reference, and programmers are left to make uninformed semantic decisions and unable to test thoroughly for correct functioning.

Under such conditions, engineers cannot have justifiable confidence in the validity of a method. Yet, as Knight has observed [19], tools are increasingly used in the design of safety critical systems. Such tools should be treated as critical engineering components. He has cited numerous, serious lapses in engineering processes with respect to tools and methods. Nevertheless, the use of poorly grounded methods and tools appears widespread, putting enterprises and the public at risk.

For example, in 1996 the United States Nuclear Regulatory Commission issued an alert [23] to all operators and builders of nuclear power plants, warning of significant errors in several tools used in nuclear reactor design and analysis. Hatton and Roberts' study of seismic analysis tools showed that they produced different results even though ostensibly computing the same function [15]. A study of reliability tools by Amari et al. revealed common errors in their analysis algorithms [3].

As domain experts increasingly develop modeling and analysis methods and deploy them in tools, it is incumbent on the software engineering research community to highlight risks and impediments, and to provide approaches to mitigate and overcome them. First and foremost are the risks presented by inadequate semantic specification of methods used in critical design situations. Until these risks are addressed, our inability to develop easy-to-use tools at low cost can be viewed as a positive safety mechanism—but far from ideal. In this paper, we show it is possible to mitigate the risks substantially and cost-effectively, and, having done that, to deploy methods into industrial use through usable tools. We now sketch the approach and present our evaluation.

## 3. Approach and basic results

Several authors have noted that significant progress in software engineering will occur as the profession becomes increasingly specialized [17,25]. By identifying and developing practices that exploit characteristics of particular application domains, software researchers can provide developers with the means to create better software than that produced with general-purpose development methods.

A key element of our approach to tool development is based on an observation by Shaw [25]: Most applications devote less than 10% of their code to the overt function of the system—in this case analyzing models. 90% is devoted to support functionality such as text and graphical editing, data validation, etc. This means that the bulk of the development effort for a sophisticated tool is spent on a large *superstructure*, in support of a small *core*.

Our approach synergistically combines two ideas [6]. First, we address the soundness of the method and trustworthiness of the core through selective, narrowly targeted use of formal methods. Second, we provide full-featured, easy-to-learn and use superstructure at low cost by using suites of mass-market packages as components, in a style we have called package-oriented programming.

### 3.1. Formal methods for modeling and analysis

The first aspect of our approach is the targeted use of formal methods to define and validate of the syntax and semantics of the modeling language [9]. The savings garnered by package-oriented design can be enough to free significant resources for such applications of formal methods. We limit the scope to the *core*. It is small in relation to the tool, which limits costs, and its soundness is so essential that we expect formalizing and validating it to have disproportionate benefits.

**3.1.1 Formalizing modeling languages.** At the heart of our evaluation of the technical and cost effectiveness of formal methods in this domain is a specification, in Z [27], of the abstract syntax and semantics of a language for dynamic fault tree (DFT) analysis of fault-tolerant systems [5,11]. Dynamic fault trees extend static fault trees [31], originally developed for reliability analysis of the Minuteman missile system [32].

A *fault tree* is a rooted, directed graph with *basic events* as leaves and *gates* internally. Basic events model component failure events; gates, subsystem failure events; and the root, a system failure event. Given failure statistics for basic events, a probability of system failure is computable. Static fault trees have combinatorial gates (e.g., *and*, *or*), to model how event *combinations* lead to failures. In fault tolerant systems, *order* can matter. Dynamic fault trees thus include order-sensitive gates. Interactions amongst these gates and other modeling features complicates the semantics of DFT's to the point that the use of formal methods appears to be indicated.

An early non-formalized DFT language was implemented in Dugan's prototype DIFtree tool [14]. A revised version, also not formalized, is at the core of our deployed, package-based DFT tool, Galileo [7,29,30]. Our formal specification is the basis for a new tool, Nova.

The specification is 55 double-spaced pages of Z: about 100 schemas and axioms, structured in a denotational style, separately specifying abstract DFT syntax and the mapping of DFT's, through an intermediate semantic domain (*failure automata*), to Markov chains (MC's). Given a DFT, its MC encodes sequences of failure events. States in which the root event has occurred denote system failures. The analysis result is the likelihood of being in such a state. The mapping from MC's to probabilities is understood. The issue was the translation of DFT's to MC's. That is what the specification defines.

We developed an initial version of the specification, which we informally validated by technical reviews conducted with domain experts. We also subjected the specification to a limited formal validation. These activities provided significant benefits, creating opportunities to improve the language and existing tools. For example, no one had ever stated clearly whether failures could occur simultaneously; the language was non-orthogonal; and existing implementations had significant faults [9].

These results are not surprising. Formal methods have been used to discover and clarify complex systems many times. The question we investigated was rather that of the *cost-effectiveness* of developing and validating such specifications in collaboration with domain experts in a low-budget setting. We found that the initial development of the specification and its informal validation was cost effective and technically productive. Initial development of the specification took the authors about four months of part-time effort. Informal validation involved a sequence of about ten weekly two-hour meetings with domain experts, in which we explained, discussed, and, as needed, revised each line of specification.

**3.1.2 Formal validation.** We also evaluated tool-assisted formal validation, applying several tools to our specification: fuzz [26], ztc [18], and Z/Eves [24]. The first two check syntax; the third is an engineer-assisted theorem prover. On the *benefits* side, Fuzz and ztc were inexpensive to use and useful, finding both syntactic and conceptual errors. However, syntax checks are clearly not adequate to provide strong validation.

Theorem proving provides iron-clad validation of claims that the constructs denoted by a specification have stated properties. We proved two types of theorems: domain checks to show that no function is applied outside its domain, and domain-specific theorems formulated during development of the specification.

Our theorem proving revealed three specification errors. Two involved our idiosyncratic approach to specifying real-number computations in Z. Z does not represent real numbers; but we wanted to capture core mathematical formulations, which get translated directly into code. We did this by specifying real numbers as a given type

and specifying standard operations on this type, abstracting away all meaningful axioms about the *real* real numbers. The errors here were not consequential.

The third error we found was non-trivial. It involved a feature of the DFT language called *coverage modeling*. Coverage modeling allows the engineer to represent three possible outcomes of a component failure: it might not propagate at all (masked transient); it might propagate normally; or it might produce an immediate system-level failure (due to a catastrophic component failure, such as an explosion). During validation, we found a fault in a failure rate computation for coverage parameters. In addition, this led us to an error in Galileo's implementation, where probability of masked failure was not correctly computed as  $1 - P_{\text{single point failure}} - P_{\text{covered failure}}$ . The bottom line: We did find an error and are now more confident in the specification with respect to the theorems we proved, and, to some extent, beyond.

The problem we encountered with formal validation was its *cost*. We encountered three difficulties. First, the Z/Eves tool itself suffered from some of the problems we are addressing in this paper. It lacked features and had usability problems that made it costly to use. The interface was idiosyncratic and did not follow standard conventions. For example, the *File* menu disappears when the user edits a proof, and the *.zev* file extension is not automatically appended to file names when necessary. More seriously, the user has to run prior proofs before starting new ones, and each proof must be manually invoked in turn. Our validation has slightly over 100 proofs. While such problems are not inherent, they were a practical and significant productivity obstacle.

Second, the tool was computationally expensive to use. Syntax checking all paragraphs and proving all theorems required about two hours compute time on a 1.2 GHz PC. When working on proofs near the end of the specification, we dreaded having to change an early part, requiring manual re-checks of intervening paragraphs that might have depended on the changed paragraph.

Third, and perhaps most seriously, using the tool required significant expertise to formulate proof scripts. We found the prover to be robust and powerful, but we were often forced to seek guidance from its author for all but the simplest proofs. In more than one case we could not have continued without his expert guidance.

In the end, the cost of theorem proving was high enough to be nearly prohibitive *for our purposes*: given a small budget, competing demands, and hard-to-reach benefits. This is not to say the cost is too high for other purposes. However, cost did curtail our attempts to prove anything beyond elementary theorems about the formal system we constructed.

### 3.2. Package-oriented programming

For several years we have been evaluating the use of architecturally coherent mass-market packages as components for low cost tool superstructures. The approach, *package-oriented programming* (POP) [8,30], exploits technologies developed by industry to support tasks such as document embedding and package scripting. POP leverages the vast investments made in these tools and the economies of volume pricing. Users benefit from careful usability engineering, rich functionality, familiarity, and rich interoperability—all at very low cost.

Our evaluation is based on the use of POP to develop Galileo [7,12,29]. The ongoing development of this tool is based on feedback from industry, and especially on NASA requirements. By evaluating the results for *industrial viability* we aim to avoid false positives generated against synthetic or less demanding requirements.

It was clear that such packages could be integrated and programmed to some degree. What was not clear was whether they could be used to build real, commercially competitive tools at low cost. In earlier work, we reported that the approach is promising but risky [8]. For example, more than once we had to bend requirements to make designs work. What we now report is that the approach has enabled us to deliver the intended tool with real industrial acceptance. Section 6 presents the results of two surveys of end users who have used Galileo, and discusses adoption of the tool.

### 3.3. The combined approach

The approach we present combines the formal and package-based aspects in a technically and economically synergistic manner. The component-based aspect provides usability and superstructure functions, greatly reducing the cost to develop, learn, and use the software. On the other hand, the criticality and complexity of the modeling method impels us to consider using formal methods. We leverage the savings produced by the package-based approach to focus more resources on the formalization and validation of the relatively small but crucial modeling method and language. We recognize the risk that mass-market packages might somehow corrupt data passing through them and thereby output incorrect results, but we do not believe this risk to be dominant.

## 4. Case study: reliability engineering

As we have said, the application domain for our work is that of reliability modeling and analysis of fault-tolerant computer-based systems. In this domain, models represent failure phenomena in fault-tolerant systems with complex redundancy management. Models are ana-

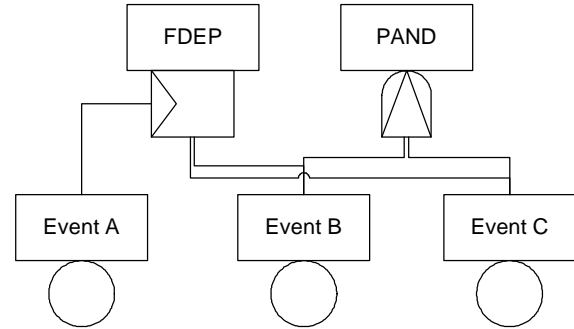


Figure 1: A small dynamic fault tree

lyzed to provide estimations of key properties, mainly unreliability. Analysis results are typically used in design to help engineers meet reliability requirements effectively and economically. Our case study in tool development centers on dynamic fault tree analysis. Central to this method are languages for expressing dynamic fault trees.

### 4.1. The complex and subtle DFT language

Dynamic fault trees are more expressive than static trees, as many fault-tolerance mechanisms are sensitive to order. For example, the use of a cold spare component depends on the prior failure of the primary. Unfortunately, the addition of order-dependent semantics complicates the straightforward semantics of static fault trees.

Prior to our work, the semantics of the DFT language had been expressed only in an ad hoc way: with informal prose, isolated examples, and prototype programs [4,5,11,14]. Unfortunately, these methods are inadequate for specifying the semantics of complex languages. Informal prose descriptions are incomplete and ambiguous. Definitions for individual cases do not capture the general case. Source code implementations are precise but resistant to human understanding and validation. Moreover, in the absence of a high-level specification there is no basis for rigorous verification [9].

Consider an example DFT. See Figure 1. *PAND* is a priority-AND gate. The failure event corresponding to this gate occurs if the input events, *Event B* and *Event C*, occur in that order. *FDEP* is a functional dependence, stating that if the trigger event, *Event A*, occurs, the dependent events, *Event B* and *Event C*, occur.

This example highlighted a semantic ambiguity in the original language. Does *Event A* cause the *simultaneous* occurrence of the dependent events, or not? The original, informal semantics of the priority-AND gate did not address the issue of simultaneous failure of the inputs. If the ordering is strict, then the *PAND* event should not occur if the inputs occur simultaneously. If the ordering is not strict, the *PAND* event should occur in this case.

This kind of ambiguity is not just an academic curiosity. An engineer at Lockheed-Martin built this tree while modeling a system in an attempt to understand the semantics of the language as implemented in DIFtree. Casual explanations of the modeling approach did not address this case. Moreover, the implementation at the time was later found to answer the question inconsistently at different points in the program [9].

## 4.2. Inadequate tool support

Prior to our work, DFT modeling tools lacked key functionality and usability properties. Both HARP [13] and DIFtree [14] provided rudimentary graphical interfaces, and lacked many features now important in practice. DIFtree, for instance, was a successful research prototype, and served its purpose well in that capacity: to prove the worth, in principle, of a novel modeling method. However, its lack of modern features, its questionable dependability, and its being tied to Unix at a time during which reliability engineers were migrating to Windows, stood as solid impediments to the delivery of the DFT modeling method, itself, from the laboratory to practice.

## 5. Nova

In this section we present our efforts to build Nova, a demonstration of the feasibility of combining the two elements of our overall approach. Like Galileo, Nova is a tool for the construction and analysis of dynamic fault tree models. Compared to Galileo, Nova is an advanced prototype tool with several unique properties. First, the DFT language that it supports is a revised version based on our formal specification. Second, the implementation of the editing interface is based on more aggressive specialization of the POP components. Fault tree editing operations are implemented using components' scripting languages, rather than in separately coded procedures. Third, the analysis engine is a new implementation based on the formal semantics we have defined.

We are addressing the question of the dependability of the Nova and Galileo core implementations in work that is beyond the scope of this paper. Here, we focus on the POP-based user interface.

Figure 2 presents the Nova interface. It consists of a Word-based textual editor, a Visio-based graphical editor, and an Excel-based basic event editor. These products are parts of the Microsoft Office suite: Word is a text editor; Visio, a drawing editor; Excel, a spreadsheet. For the development of the interface, we took the opportunity to explore aggressive specialization of the POP components. For example, the figure illustrates the automatic syntax highlighting of keywords in the textual editor. For

the sake of brevity, we describe only the more advanced Visio-based graphical interface.

Visio provides general graphical editing functionality such as zooming, scrolling, formatting, saving, printing. In addition, we utilized its *UpdateUI* interface to specialize our interface in several dimensions to support the editing of DFT's in their concrete graphical form.

First, we created a "stencil" of shapes for graphical depictions of the DFT modeling constructs. These shapes have dynamic behavior. For example, connectors automatically "hop" over each other, shapes move automatically to prevent overlap, and text boxes expand to accommodate long label names. Some of these features were supported natively by Visio, and others were implemented using the package's shape design capabilities.

Second, a menu and toolbar of functions has been added to perform DFT-specific operations such as changing the type of a gate or selecting a subtree. In order to implement these domain-specific operations, we took advantage of Visio's built-in macro creation capabilities. For example, consider the "change shape type" functionality. The code first checks that a shape is selected. It then displays a dialog box with the names of the various shape types. After the user makes a selection, any connections to the shape are saved and the original shape is deleted. Then the new shape is created and moved to the same location. Finally, the prior connections to the original shape are established for the new shape. By automating this fault-tree-specific editing operation, we relieve the user of having to perform each of these operations manually.

Our third type of customization was to remove or replace inappropriate Visio-native functionality. For example, Visio supports the notion of shape-to-shape connections in which the closest connection point is automatically selected. Since the DFT language makes a distinction between input and output connection points, the editor enforces a connection-point-to-connection-point policy. To do this, it detects when new connections are made by trapping the connection event raised by Visio. The editor then analyzes the type of connection, and either automatically determines the proper connection points in unambiguous cases, or queries the user otherwise.

Lastly, we enhanced the behavior of Visio to ease the task of building fault trees. For example, we found during the Galileo workshops that users often had trouble determining if both ends of a connector were properly attached. To help with this problem, we decided to implement a feature in Nova so that the connector is dashed when one end is not connected, and solid if both ends are properly connected. Implementing this feature was easy—it took only about an hour due to the programmability of the Visio package.

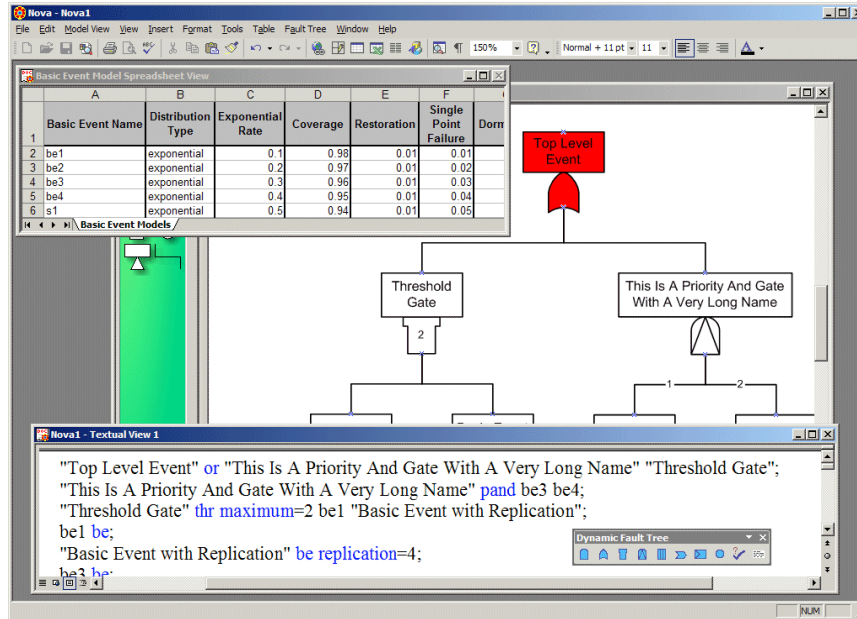


Figure 2: A screenshot of Nova

The Nova tool demonstrates the feasibility of our combined approach. It is a POP-based tool for engineering modeling and analysis implementing a modeling language having a formal, reasonably validated semantics.

## 6. End-user evaluation of Galileo

On the basis of the potential demonstrated by early prototype versions of Galileo, NASA Langley Research Center funded the development of a version called Galileo/ASSAP. The new tool version was required to support several new modeling and analysis constructs and to be usable in practice. It has been featured in three NASA workshops on dynamic fault-tree modeling. Engineers and managers from several NASA divisions were present at the first workshop. The second and third involved engineers from the space station and space shuttle projects.

### 6.1. Survey objectives and design

The workshops provided a unique opportunity to assess end user perceptions of Galileo/ASSAP. To that end, we developed two surveys and asked participants to take them. Participation was optional. To increase participation, we created a short survey with 34 key questions, and a longer survey with 77 more in-depth questions. Most of the questions were multiple-choice, with a few short-answer and ranking questions. Participants were given the opportunity to comment on every question in order to clarify their answers or give additional information. Questions were designed according to the guidelines sug-

gested by Dillman [10]. Every effort was made to limit the range of interpretation of the questions and to reduce bias.

The overall goal of the surveys was to evaluate user perceptions of the tool in terms of usability and features. To that end, the surveys contained questions about the difficulty of performing common tasks with the tool. There were also several questions related to the user's impressions of a tool built using mass-market applications as components. Because we expected a small number of respondents and moderate variation in experience and skills, the surveys also included a number of questions designed to help us interpret the answers relative to the skills and experience of the respondents.

In order to help assess the ability of the POP approach to deliver an industrially viable tool, we also asked several questions which compares Galileo to well-established commercial tools. Questions were included to assess the users' impressions of tools that they frequently use, and to compare Galileo's features and usability.

### 6.2. Results

Sixteen engineers from twelve NASA groups, including NASA contractors, answered both surveys. The majority of the engineers considered themselves to be familiar with reliability modeling and analysis techniques, with five engineers using modeling and analysis tools every day. Almost half of the respondents analyze systems whose failure could lead to over US \$1B lost and/or loss of life.

The two key requirements which we identified were confirmed by the surveys—the users cited “an easy-to-use user interface” and “accurate and precise analysis results” as being the two most important characteristics of a tool—above other options such as support for a range of modeling capabilities and speed. Users also confirmed our claim that dependability is crucial—a formal specification of the modeling language was second only to a comprehensive test suite as a means for increasing trust.

In terms of usability, most respondents indicated that the Word- and Visio-based views were easy to use. Compared to other tools, one said the usability was much worse, while all others said it was the same or better.

To understand user perception of Galileo features, we asked about two types of editing functionality: the functionality provided natively by the packages, and the domain-specific editing capability that we implemented by specializing the packages. The majority of users said they were satisfied with the general editing functionality, and nearly all said that the domain-specific operations met their needs well. All said the editing capabilities of Galileo were the same or better than other tools.

We also asked the users about specific features which were designed to accommodate package constraints, and which implemented modified versions of requirements. For example, views are updated using a batch approach instead of our original goal of incremental update, driven in part by the limitations of the Visio component [30]. We found that all users were satisfied with the capabilities which we were able to implement.

While learning to use Galileo, the users were told that the tool was built using Word and Visio. We asked the users several questions about these packages. All users were familiar with Word, but almost half were using Visio for the first time. When asked if their familiarity with the packages helped them use Galileo, nearly all said it helped at least a little, and several said it helped a lot. The majority of users were also satisfied with the performance of the tool, even though it uses large packages as components and is not optimized for speed.

These survey results are encouraging. End user evaluation is a key arbiter of success in the use of POP for the construction of tools. Despite the “beta” status of Galileo, the surveys indicate that the Galileo tool meets or exceeds the expectations of engineers. In most respects the tool was judged comparable with commercial tools.

When asked what surprised them the most about the tool, several users cited the usability, saying “the ease of use was better than expected”, “[the] program is very user friendly”, and “very friendly user interface”. Several users liked the use of standard packages as components, saying they were surprised that it has “transparent linkage between Word and Visio”. One user went so far as to say “the reuse of Word/Visio [is] a rather brilliant idea”.

### 6.3. Adoption of Galileo by industry

Following their experiences using Galileo during the workshops, certain NASA engineers involved with the International Space Station (ISS) project adopted the tool. Today, the Galileo/ASSAP version of Galileo is used by the space station’s fault diagnosis and repair group to model the causes of observed failures, but not for estimating failure probabilities. According to feedback from the engineers, the tool’s editing interface provides capabilities that exceed those of the tools that they had been using.

In fact, the engineers have reported that Galileo’s ease-of-use has led to a significant change in their practice. Previous tools required domain experts to work with reliability engineers to develop models. With Galileo, domain experts are able to model the system themselves, without having to depend on other engineers.

NASA’s satisfaction with the Galileo/ASSAP tool has also led to a request for a follow-on version. This version is planned to include new features at the request of both NASA Langley Research Center (primary sponsor) and the ISS group at NASA Johnson Space Center.

The adoption of Galileo by NASA and the desire to extend the product are good indicators that the POP approach has succeeded in delivering the tool capabilities that real users value. Galileo was developed by a small team in an academic setting, and yet has features and usability that rival commercial tools and that appear able to satisfy the needs of major industrial organizations.

## 7. Evaluation

In earlier work, we presented less mature and separate evaluations of the constituent efforts. Previous work showed that the POP approach was promising but subject to risks, known and unknown. The survey data and industrial acceptance we present in this paper provide end user confirmation that the POP approach is capable of delivering tools with industrially viable feature sets and usability.

In terms of formal methods, we have shown that, with modest effort and basic mathematical and domain knowledge, it is possible to identify and eliminate serious semantic and implementation errors [9]. However, our experience was that the cost of formal validation was high enough to be inconsistent with a small budget.

Nova demonstrates the technical feasibility and cost-effectiveness of the combined approach. In less than two person-years, with the effort of a graduate student, advisor, and domain expert, we built a tool with usability better than Galileo’s and having a formal foundation for trust.

In terms of lines of code, Nova is implemented in just under 30,000 lines of commented code. This count includes 3,100 lines of code specializing Word, 8,800 lines in specializing Visio, 9,000 lines for the analysis engine, and 5,700 lines for overall application control.

Nova has not yet been evaluated by end users, but we are confident that its interface will be as well-received as Galileo's. Nova employs more aggressive use of the POP components, resulting in better responsiveness and more sophisticated behaviors and editing operations.

## 8. Related work

In this section we describe three areas of related work: package-oriented programming, applied formal methods, and tool development methods.

### 8.1. Package-oriented programming

Lédeczi et al. [21] describe a COTS-based design environment generator exploiting packages as components. Given a meta-model of a language, their tool generates an environment. Our work focuses on the integration of applications as components. Theirs deals more with environment generation issues.

Succi et al. [28] are investigating the integration of POP components. However, they target cross-platform integration using a Java-based architecture as the integration mechanism. In contrast to our work, they do not attempt to achieve tight functional integration, and do not address user interface integration.

### 8.2. Applied formal methods

Efforts are underway to formalize modeling languages for software engineering, including architecture [1] and connectors [2]. The goal is to put such languages on mathematically sound foundations, enabling semantically well defined analysis methods. For example, architectural checks can assess compatibility of connectors and components using methods similar to type checking. This work supports our hypotheses about the technical value of formal methods. However, our work is more than technical. We seek to understand the benefits of formality relative to the costs. The question is, what techniques create value net of their costs, considering competing demands, and under demanding constraints?

Our experience using Z/Eves is similar to that of Knight et al. [20], who used the PVS theorem prover on a modest-sized nuclear power plant specification. They too found it hard to formulate theorems and proof strategies, and were hindered by poor tool usability.

### 8.3. Tool development methods

Related to our work are techniques for building general modeling and analysis environments. Examples include the Generic Modeling Environment [21], MetaEdit+ [22], and DOME [16]. The idea is to develop reusable frameworks that can be used to instantiate new tools by specifying the aspects specific to the application domain. Our work is distinct in several dimensions. First, our strategy is to address the components used to construct tools, as opposed to an overall reusable framework. In this respect our work is not incompatible with the framework approach—it is possible to use POP components within a generic framework. Second, developers of reusable frameworks are faced with the challenge of not only providing a wealth of functionality and high usability, but also making this functionality easily reusable. Lastly, many modeling languages (including DFT's) appear too complex to capture easily using the hierarchical and constraint-based semantics used by generic frameworks such as GME.

## 9. Conclusion

In this paper we presented and evaluated—largely in economic terms—an approach to developing sound modeling methods supported by functionally rich and easy to use software tools. Using packages as components enabled the development of an industrially effective tool at low cost, with enthusiastic acceptance by NASA engineers as evidence of acceptability. Using formal methods fundamentally improved the soundness of a complex method. The Nova tool combines these components, hosting a formally specified and validated language in an improved package-based tool. Nova demonstrates, apparently for the first time, that it is possible to develop sound methods and make them readily available for industrial use at a cost that is modest by any reasonable measure. It took less than two person years to develop the specification, implement it, and deliver it in a package-based tool.

The cost-effective verification of implementations of analysis methods is a remaining key issue, but one beyond the scope of this paper. A solution would largely complete a three-part approach to the construction of sound methods and effective tools for engineering. We have identified, are now evaluating, and will describe in a forthcoming paper, a powerful testing technique leveraging our investment in formal specification.

This work has the potential to have a significant impact on engineering practice in at least two dimensions. First, by dramatically reducing the cost barriers to developing genuinely usable modeling methods, it promises to catalyze the development of such methods and their transiting from laboratories into practice.

Second, we see that using complex modeling methods in critical industrial, governmental, or military engineering activities without software dependability assurances based on formal foundations is fraught with risk. This paper shows that well known formal methods and a willingness to work with domain experts, although not a *silver bullet*, are enough to significantly and cost-effectively contribute to the validation, implementation and documentation of modeling methods. It is thus reasonable to start to recognize and seriously question the use of computerized modeling methods lacking documented formal semantic foundations for dependability.

## Acknowledgements

This work was supported in large part by the National Science Foundation under grant ITR-0086003. We thank colleagues at NASA Langley Research Center, Johnson Space Center, and across the organization for agreeing to participate in our survey. We also warmly thank our collaborators in reliability and software engineering research, including Joanne Bechta Dugan and John Knight, for their support. Finally, we thank all of the anonymous reviewers for their constructive critiques.

## References

- [1] Gregory D. Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–64, October 1995.
- [2] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–49, July 1997.
- [3] Suprasad Amari, Joanne Bechta Dugan, and Ravindra Misra. A separable method for incorporating imperfect coverage into combinatorial models. *IEEE Transactions on Reliability*, 48(3):267–74, September 1999.
- [4] Anju Anand and Arun K. Somani. Hierarchical analysis of fault trees with dependencies, using decomposition. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 64–70, Anaheim, CA, 19–22 January 1998.
- [5] Mark A. Boyd. Dynamic Fault Tree Models: Techniques for Analysis of Advanced Fault Tolerant Computer Systems. PhD thesis, Duke University, Department of Computer Science, April 1991.
- [6] David Coppit. *Engineering Modeling and Analysis: Sound Methods and Effective Tools*. PhD thesis, The University of Virginia, Charlottesville, Virginia, January 2003.
- [7] David Coppit and Kevin J. Sullivan. Galileo: A tool built from mass-market applications. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 750–3, Limerick, Ireland, 4–11 June 2000. IEEE.
- [8] David Coppit and Kevin J. Sullivan. Multiple mass-market applications as components. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 273–82, Limerick, Ireland, 4–11 June 2000. IEEE.
- [9] David Coppit, Kevin J. Sullivan, and Joanne Bechta Dugan. Formal semantics of models for computational engineering: A case study on dynamic fault trees. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 270–282, San Jose, California, 8–11 October 2000. IEEE.
- [10] Don A. Dillman. *Mail and Internet Surveys: The Tailored Design Method*. John Wiley & Sons, 2nd edition, 1999.
- [11] Joanne Bechta Dugan, Salvatore Bavuso, and Mark Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability*, 41(3):363–77, September 1992.
- [12] Joanne Bechta Dugan, Kevin J. Sullivan, and David Coppit. Developing a low-cost, high-quality software tool for dynamic fault tree analysis. *Transactions on Reliability*, 49(1):49–59, March 2000.
- [13] Joanne Bechta Dugan, Kishor S. Trivedi, Mark K. Smotherman, and Robert M. Geist. The hybrid automated reliability predictor. *Journal of Guidance, Control, and Dynamics*, 9(3):319–31, June 1986.
- [14] Joanne Bechta Dugan, Bharath Venkataraman, and Rohit Gulati. DIFTree: A software package for the analysis of dynamic fault tree models. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 64–70, Philadelphia, PA, 13–16 January 1997.
- [15] Les Hatton and Andy Roberts. How accurate is scientific software? *IEEE Transactions on Software Engineering*, 2(10):785–797, 1994.
- [16] Honeywell. DOME users’ guide. URL: <http://www.htc-honeywell.com/dome/support.htm>.
- [17] Michael Jackson. Problems, methods and specialisation. *Software Engineering Journal*, 9(6):249–55, November 1994.
- [18] Xiaoping Jia. *ZTC: A type checker for Z. Notation user’s guide*. URL: <http://se.cs.depaul.edu/fm/ztc.html>.
- [19] J. C. Knight. Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, pages 547–9, Orlando, Florida, 19–25 May 2002. IEEE.
- [20] J. C. Knight, Colleen L. DeJong, Matthew S. Gibble, and Luis G. Nakano. Why are formal methods not used more widely? Fourth NASA Formal Methods Workshop, Hampton, Virginia, September 1997.

- [21] Akos Ledeczki, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing*, Budapest, Hungary, 17 May 2001.
- [22] MetaCase Consulting. Domain-specific modeling: 10 times faster than UML. URL: <http://www.metacase.com/papers/-index.html>.
- [23] Office of Nuclear Reactor Regulation. Requirements in 10 CFR part 21 for reporting and evaluating software errors. Technical Report NRC Information Notice 96-29, United States Nuclear Regulatory Commission, 20 May 1996.
- [24] Mark Saaltink. The Z/EVES system. In *ZUM '97: Z Formal Specification Notation. 11<sup>th</sup> International Conference of Z Users. Proceedings*, pages 72–85, Berlin, Germany, 3–4 April 1997. Springer-Verlag.
- [25] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 7(6):15-24, November 1990.
- [26] J. M. Spivey. *The fuzz manual*. URL: <http://spivey.oriel-ox.ac.uk/~mike/fuzz/>
- [27] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
- [28] Giancarlo Succi, Witold Pedrycz, Eric Liu, and Jason Yip. Package-oriented software engineering: a generic architecture. *IT Professional*, 3(2):29-36, March-April 2001.
- [29] Kevin J. Sullivan, Joanne Bechta Dugan, and David Coppit. The Galileo fault tree analysis tool. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 232-5, Madison, Wisconsin, 15-18 June 1999. IEEE.
- [30] K. J. Sullivan and J.C. Knight, “Experience Assessing an Architectural Approach to Large-Scale, Systematic Reuse,” Proceedings of the 18th International Conference on Software Engineering, Berlin, March 1996, pages 220-229.
- [31] W. E. Veseley, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. U. S. Nuclear Regulatory Commission, NUREG-0492, Washington DC, 1981.
- [32] H. A. Watson and Bell Telephone Laboratories. Launch control safety study. Technical report, Bell Telephone Laboratories, Murray Hill, NJ, 1961.