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
+1 804 982 2206
sullivan@cs.virginia.edu

Abstract

Modeling is indispensable in engineering. Safe, effective modeling ***methods*** *require* ***languages*** *having clearly specified and validated semantics, and low-cost, feature-rich, easy-to-use software* ***tools****. Today we lack cost-effective means to develop such methods, with serious consequences for engineering. We present and evaluate an approach combining two techniques: formal methods to aid in language design and validation; and package-oriented programming for effective tools at low cost. We have evaluated the approach in an end-to-end feasibility experiment. First, we deployed an existing language for reliability analysis to NASA in a package-oriented tool and surveyed NASA engineers to assess its industrial effectiveness. Second, we designed a formally specified (and significantly corrected and improved) modeling language. Finally, to assess the overall effort required, we developed a package-based tool from scratch which embodies the new language. The data support the claim that the approach promises to enable cost-effective deployment of sound methods by effective software tools.*

1. Introduction

Modeling and analysis *methods* are at the heart of engineering design. Any such method provides the modeler with a modeling *language* used to describe systems, the semantics of which are in a mapping of expressions (i.e., models) to estimates of system properties. Functions for creating and manipulating models, and implementations of such mappings, are typically supported by software *tools*.

For a modeling method to be used safely and effectively, it must be semantically sound independent of its implementation in any particular tool, and supported by a high-quality tool. Semantic soundness demands a clear, abstract, precise, compete, general and validated definition of the mapping from models to results. Tool quality demands both usability and reliability. Usability requires the provision of a broad set of easy-to-learn and easy-to-use functions needed *in engineering practice*—functions that go well beyond bare-bones model editing and analysis, to include such things as printing large models on engineering-size paper, cut-and-paste of models into presentation tools, and sophisticated graphical editing. Non-computer scientists have become accustomed to features and usability on par with mass-market packages; and engineers now rightly demand this level of usability even in highly specialized modeling tools. Reliability, on the other hand, demands, first and foremost, that a tool implement the semantics of the modeling language faithfully.

The problem we address in this paper is that the safe and effective use of modeling and analysis in engineering is impeded by unsolved problems in software engineering. Today we are largely unable to deliver, at low cost, semantically sound methods supported by high quality tools. First, we lack *proven cost-effective* approaches to ensuring the semantic soundness of methods too semantically complex to succumb to casual and informal specification. Second, we lack *low-cost* approaches to producing high quality tools for specialized engineering markets.

The contribution of this paper is the presentation and end-to-end experimental evaluation an approach to this problem. The approach combines well known formal methods for semantic specification [8,9] with a package-oriented approach [31] to component-based tool design. Savings enabled by package-oriented design enable resources to be devoted to semantic soundness. By end-to-end, we mean that our experiment includes deployment and evaluation of a package-based tool into a production setting (NASA, International Space Station project), and the development of a new tool, Nova, combining the package- and semantical methods. Our data suggest that it is possible, at a cost that is modest by any reasonable

measure, to deliver *sound* modeling methods into industrial use supported by high-quality software *tools*.

In the next section we describe our approach in detail. Section 2 discusses the foundational role that specification must play in the development and deployment of modeling methods. Section 3 describes the approach and basic results. Section 4 introduces the application domain in which we have evaluated our work. Section 5 describes our efforts to build the Nova tool using the combined approach. Section 6 presents results from the end-user evaluation of Galileo. Section 7 evaluates our work, Section 8 describes related work, and Section 9 concludes.

2. Semantic Foundations of Trustworthiness

The essence of a modeling method is in the definition of the mapping from system models to analysis results. This mapping is ideally expressed in both specification and implementation forms. Of the two, the specification is clearly fundamental. It provides the basis for domain experts to validate the method, for programmers to implement it, and for users to understand and use it.

Lack of an adequate specification complicates validation by denying domain experts a definition of the method in all of its complexity. Users are without the basis for a definitive reference. And programmers, without a clear definition of precisely what the program has to do, are placed in the position of making uninformed but semantically deep decisions about the essence of the modeling method, and are unable to test against clear statements of correct functioning.

In such cases, it is impossible for engineers using a method to have justifiable confidence in its validity, in the validity of a model relative to the system modeled, or in the validity of results produced by computer analysis of the model. Anecdotal evidence suggests the use of such methods and tools today is widespread and that it puts enterprises, and in some cases, the public, at risk.

Knight, in particular, has observed [20] that software tools are increasingly used in the development of safety critical systems, and that this role requires the software to be treated as a critical component of the overall engineering process, and he has cited numerous instances of serious lapses in engineering processes with respect to tools and the methods they implement.

In 1996 the United States Nuclear Regulatory Commission issued an alert [24] 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' analysis of seismic analysis tools showed that they produced different results even when ostensibly computing the same function [16]. The analysis by Amari et al. of several reliability tools revealed the same error in their analysis algorithms [3].

As domain experts increasingly envision and develop complex modeling and analysis methods and attempt to deploy them in software tools, it is incumbent on the software engineering research community to help build an awareness of the risks and impediments, and to provide approaches to mitigate and overcome them. First and foremost are the risks presented by inadequate semantic specification of modeling methods used in critical design situations. Until these risks are addressed, our inability to develop highly usable method-deploying tools at low cost can be viewed as a positive safety mechanism—but far from ideal. In this paper, we identify the risks, show that it is possible to mitigate them both substantially and cost-effectively, and, having done that, to deploy methods into industrial use in the form of highly usable software tools. We now sketch the approach and discuss its use in one end-to-end method-and-tool-development experiment.

3. Approach and basic results

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

3.1. Package-oriented programming

Several authors have noted that significant progress in software engineering will occur as the profession becomes increasingly specialized [18,26]. By identifying and developing practices that exploit characteristics of particular application domains, software engineering researchers can provide developers with the means to create better software than that produced using general-purpose software development methods.

Our approach to developing a tool, given a definition of a method and language (often graphical), is based on several observations. The first, made by Shaw [26], is that most applications devote less than 10% of their code to the overt function of the system—in this case the analysis of models. 90% of the code is devoted to support functionality such as text editing, graphical editing, data validation, etc. In other words, much of the development effort required for a sophisticated tool is devoted to the *superstructure* which supports the *core*.

For several years we have been exploring the use of architecturally compatible mass-market packages as components to provide the bulk of the superstructure at a greatly reduced cost. This approach, which we call package-oriented programming (POP) [7,31], exploits technologies originally developed by industry to support tasks such as document embedding and package scripting. In

this design approach, multiple packages are specialized and tightly integrated at both the API and user interface levels. By using packages as massive components, POP exploits the vast investments that have been spent in their design, construction, and refinement, and the tremendous economies obtained by the volume pricing of mass-market software. In particular, users benefit from careful usability engineering, rich functionality, software familiarity, and rich interoperability—all at very low cost.

The evaluation of the approach that we present here is based on its use in the end-to-end (concept-to-industrial-use) development of Galileo [6,12,30]. Galileo is a richly functional and easy-to-use tool for reliability modeling and analysis. The development of the tool was guided by feedback from industrial organizations such as Lockheed-Martin, and recent versions have been designed to written requirements provided by NASA. We believe that by targeting industrial viability, we can better avoid a false positive evaluation of POP that might result from applying the approach to less sophisticated systems.

It's clear that packages can be integrated and programmed to some extent. What is not clear is whether it is possible to build modeling and analysis tools using these techniques which are competitive with the best commercial tools. In earlier work, we reported that the approach offers great promise but that it is also fraught with risk, mainly integration risk [7]. The unpredictable limitations of the components forced us into a strongly risk driven development model, and more than once we had to bend requirements to find workable designs. What we now report is that the approach has enabled us to deliver a tool for which there is real industrial demand and acceptance. Section 6 presents the results of two surveys of end users who have used Galileo, and discusses adoption of the tool.

3.2. Formal methods for modeling and analysis

The second aspect of our approach involves the judicious use of formal methods for the definition, validation, and revision of the syntax and semantics of modeling languages [8,9]. The savings garnered by package-oriented design can be enough to permit significant resources to be devoted to specification. We also limit the scope of this activity to the *core* method and language instead of to the overall tool. The method is "small" in relation to the tool and its soundness is so essential that we expect formalizing and validating it to have disproportionate benefits.

To evaluate the technical and cost effectiveness of this aspect of our approach we used Z [28] to formalize—and in the process to identify and correct significant problems and improve the orthogonality of—the DIFTree modeling language for dynamic fault tree analysis [5,11]. That language was originally implemented by the DIFtree tool, developed by our domain expert colleagues, and a revised version is now also implemented by Galileo.

Our specification is 55 double-spaced pages and about 100 schemas and axioms long. We structured our specification in a denotational style, separately specifying the abstract syntax of the language and the mapping of arbitrary expressions in the language through an intermediate semantic domain to the well understood domain of Markov chains. We developed an initial version of the specification, which we informally validated against the understanding of the domain experts. We then also subjected the specification to limited formal validation.

Formalizing the language and informally validating it in collaboration with domain experts revealed significant opportunities for improvement in the method, language, and existing implementations. For example, we found that neither pre-existing casual specifications nor previous tool implementations properly addressed semantically important issues such as whether failures could occur simultaneously, and if so, what the effects would be on the analysis results. We also found that modeling language to be significantly less regular and orthogonal than it needed to be, and our work led us to significant errors in existing implementations.

These results are not enormously surprising. Formal methods have been used to discover and clarify complex systems many times. The question we were investigating was primarily 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 weekly meetings of one to two hours with domain experts (Joanne Dugan and her students), during which time we explained, discussed, and as necessary, revised each line of the specification.

We are now able to report on our experience with tool-assisted *formal* validation. We applied several tools to our specification: fuzz [27], ztc [19], and Z/Eves [25]. The first two tools are syntax checkers, and the third is an engineer-assisted theorem prover.

We found formal validation using these tools to be less clearly cost-effective than our initial formalization effort and informal validation. Fuzz and ztc syntax checkers were easy to apply and useful, but syntax checks are inherently limited. Theorem proving permits validation, through formal proofs, that the constructs denoted by specifications have certain desired properties. In this respect, the Z/Eves prover did help us find deeper errors, but it was quite expensive to use. We proved two types of theorems: domain check proofs to ensure that no function is applied to a value outside its domain, and custom theo-

rems which we had formulated during initial development of the specification.

We encountered three types of problems. First, the Z/Eves tool itself suffered from the lack of features and usability problems we described earlier. The user 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. More seriously, the user must run prior proofs before starting new ones, and each proof must be manually invoked in turn. (Our specification has slightly over 100 proofs.)

Second, the tool requires a high level of expertise of the user in order to formulate workable proof scripts. We found the prover to be robust and powerful, but we were often forced to seek guidance from the author of the tool for all but the simplest proofs. In more than one case we could not have continued without his expert guidance.

Third, the prover was computationally expensive to use. Syntax checking all the paragraphs and proving all of the theorems requires about two hours of compute time on a 1.2 GHz PC. When working on proofs near the end of the specification, we would dread having to change an early part of the specification, as this would require us to manually re-check all of the intervening paragraphs.

Despite our difficulties, we did discover three significant technical errors in the specification. Two involved the computation of real-number coefficients—real numbers are a known weakness of the Z specification language, and one which we handled by creating an abstraction for reals. The third error involved a feature of the language called "coverage"—its resolution also revealed an error Galileo's analysis engine.

We are more confident in the specification with respect to the theorems we were able to prove. However, it is not clear whether we could have achieved the same results with additional informal validation. The formal validation effort did not reveal substantial errors in our understanding of the DFT language or methodology, which may indicate that the largest dividends can be had through initial formalization and informal validation.

3.3. The combined approach

The approach we present in this paper combines the formal and package-based aspects in a novel, technically and economically synergistic manner. The component-based aspect provides usability and superstructure functions, greatly reducing costs to develop, learn and use the software. The risk of producing incorrect analysis results due to the use of mass-market packages (not known for their reliability) does not appear to be dominant.

On the other hand, the critical nature of the modeling method impels us to consider the use of formal methods. We leverage the saving produced by the package-based approach to focus more resources on the formalization and validation of the relatively (logically) small yet crucial modeling method and its language.

4. Case study: reliability engineering

The application domain for our work is that of reliability modeling and analysis of fault-tolerant computer-based systems. In this domain, domain-specific languages are used to build models of systems with complex failure management. These models represent the potential failures and failure recovery behaviors of the system, and are analyzed to provide estimations of key system properties such as overall system unreliability. These analysis results are then used by the reliability engineer to assess the reliability of the system being modeled, and perhaps to modify the design to improve its reliability.

The particular modeling and analysis method which we address is dynamic fault tree analysis. Central to this method is the dynamic fault tree (DFT) language. Dynamic fault trees [5,11] are an extension of static fault trees [32], which were originally developed for analysis of the Minuteman missile system [33]. In this language, the occurrence of low-level events or the failure of basic components in a system are modeled using probability distributions. The composition of basic component failures is modeled using combinatorial or sequence-dependent constructs called gates.

4.1. The complex and subtle DFT language

Compared to static fault trees, the dynamic fault tree language is a more powerful and useful notation, as many fault-tolerance mechanisms depend upon order. For example, the use of a spare is dependent on the prior failure of the original component. Unfortunately, the addition of order-dependent semantics complicates the otherwise straightforward semantics of static fault trees.

Prior to our work, the semantics of the DFT language had been expressed only with informal prose, a few isolated examples, and prototype computer programs [4,5,11]. Unfortunately, these methods are inadequate for precisely defining complex and subtle modeling languages. Informal prose descriptions are incomplete and inherently ambiguous. Mappings of individual models to their semantics do not capture the general case. Source code and executable implementations are precise, but procedural code is resistant to human understanding and validation, and in the absence of a high-level specification there is no basis for rigorous verification [8]. The programmer may, perhaps unknowingly, make incorrect decisions about important semantic concerns.

Figure 1 presents a simple dynamic fault tree. In this model *PAND* is a priority-AND gate which fails if the inputs *Event B* and *Event C* fail in order. *FDEP* is a func-

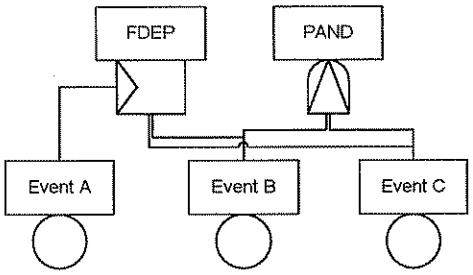

Figure 1: A small dynamic fault tree

tional dependency which indicates that the dependent events *Event B* and *Event C* fail simultaneously if the trigger event *Event A* fails.

This example demonstrates a semantic error in the language. In this case, the failure of the trigger *Event A* causes the simultaneous failure of the dependent events. However, 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 gate should not fail if the inputs fail simultaneously. If the ordering is not strict, it should fail in this case.

This ambiguity in the language semantics is not simply an academic curiosity—an engineer at Lockheed-Martin discovered it while attempting to model a system with an early version of Galileo. Unfortunately, the informal specification of the language did not address this case, and the existing implementation could not be used as a semantic reference because it did not behave consistently. Apparently the implementer did not realize the subtlety and therefore left the semantics inconsistent [8].

4.2. Inadequate tool support

Prior to our work, tools for the DFT methodology lacked the quality that users expect. HARP [13], for example, did not support a number of modeling constructs, and lacked a graphical user interface. DIFTree [14] supported a more expressive version of the DFT language, and also implemented an innovative modular solution technique. However, the graphical interface was idiosyncratic to use and lacked a number of features.

Like most research prototypes, these tools allowed researchers to explore the modeling language and provided a testbed for further research. However, the tools lacked features, were not easy to use, and were tied to Unix, which was by then not the platform of choice in engineering practice. As a result of these tool limitations, the widespread adoption of the dynamic fault tree modeling and analysis methodology appeared to be at risk.

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 work. Second, the implementation of the editing interface is based on more aggressive specialization of the POP components which implements the fault tree editing operations directly in the components. Third, the analysis engine is a new implementation based on the formal semantics we have defined.

While we recognize the importance of a verified implementation of the method, this problem is beyond the scope of our current research. As a result, Nova's analysis engine was carefully designed and implemented, but has not yet been verified correct. The details of the development of this analyzer will not be presented here. Instead, we shall focus on the POP-based user interface.

Figure 2 shows the Nova interface. It consists of a Word-based textual editor, a Visio-based graphical editor, and an Excel-based basic event model editor. For the development of the interface, we took the opportunity to explore more 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 will only describe the more advanced Visio-based graphical interface.

Visio provides the general functionality such as zooming, formatting, saving, printing, etc. In addition, we utilized Visio's *UpdateUI* interface to specialize the interface in several dimensions to support the domain-specific editing of DFTs in their concrete graphical form.

First, we created a "stencil" which contains 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 a gate's type 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" task. 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,

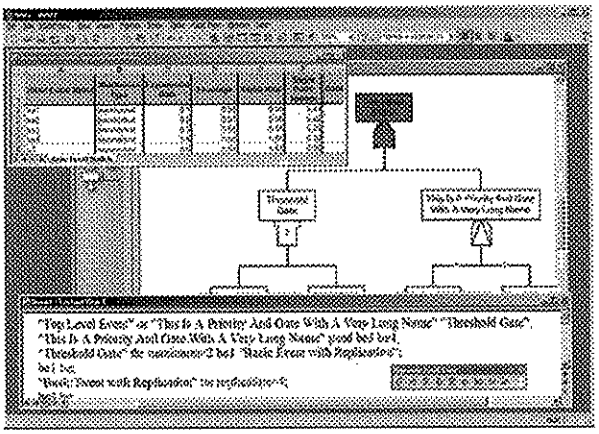

Figure 2: A screenshot of Nova

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 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 approximately an hour due to the programmability of the Visio package.

6. End-user evaluation of Galileo

On the basis of the potential demonstrated by early versions of Galileo, NASA Langley Research Center funded the development of a production version called Galileo/ASSAP. The tool was built to a rigorous set of documented requirements and testing procedures. It has been featured in three workshops on reliability modeling and analysis. Engineers from a number of NASA divisions were present at the first workshop, while the second and third workshops involved engineers from the space station and space shuttle projects.

These workshops provided us with a unique opportunity to assess end user impressions of the Galileo tool. To that end, we developed two surveys which we distributed to workshop participants.

6.1. Survey objectives and design

Survey participation in the workshops was optional. In order to increase participation, we created two surveys: a short survey with 34 essential questions, and a longer survey with 77 in-depth questions. The bulk 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 given additional information. Questions were designed according to the guidelines suggested 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 understand user impressions of the tool in terms of its usability and features. To that end, the surveys contained a number of 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 "calibrate" the answers in a subjective manner.

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 commercially available tools. Questions were included to assess the user's impressions of tools that they frequently use, and to compare Galileo's features and usability.

6.2. Results

Sixteen engineers from twelve NASA groups and contractors answered both surveys. The majority of the engineers considered themselves to be familiar with reliability analysis techniques, with five engineers using software 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 have identified were confirmed—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 supported our position that the method is crucial—a formal specification of the

modeling language was second only to a comprehensive test suite as a means for increasing user trust.

In terms of the usability of Galileo, nearly all respondents indicated that both the Word-based textual view and the Visio-based graphical view were easy to use. Compared to other tools, one user said the usability was much worse, while all others said the usability was the same or better.

To understand user's impressions of the features offered by Galileo, we asked them about two types of editing functionality: the general editing functionality provided natively by the packages, and the domain-specific editing capability which we implemented via specialization of the packages. The majority of users indicated that they were satisfied with the general editing functionality, and nearly all respondents said that the domain-specific editing operations met their modeling needs well. All respondents said that the model 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 [31]. 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 constructed from Word and Visio. We asked the users several questions about the use of 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 users said it helped at least a little, and several said that it helped a lot. The majority of users were also satisfied with the performance of the tool, even though it used two fairly large applications as components.

These survey results are encouraging. End user evaluation is the true 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 is 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, NASA engineers involved with the International Space Station project lobbied to adopt 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. According to feedback from the engineers, the tool's fault tree editing interface provides editing capabilities which far exceed that of commercial tools which 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 reliability engineers who do not understand the space station domain.

NASA's satisfaction with the Galileo tool has also led to a request for a new version of the tool. This version is planned to include new features at the request of both NASA Langley Research Center (our 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 and characteristics that users require. Galileo was developed by a small team in an academic setting, and yet has features and usability which rival commercial tools and satisfy the needs of major industrial organizations.

7. Evaluation

In earlier work, we have presented less mature and separate evaluations of the constituent efforts. Our previous work on Galileo tool showed the POP approach was promising but still subject to risks known and not. The survey data and industrial acceptance we present in this paper provide confirmation by end users that the POP approach to developing tools such as ours is capable of delivering tools with the necessary features and usability.

One key assumption that we make is that the methods employed by engineering tools lack a suitable level of semantic trustworthiness. There are obviously many notations whose meaning is well-defined and for which formal methods are unnecessary. However, we believe that the continuing development of tools to support domain-specific modeling methods make this increasingly important. In such cases, we have shown that with modest effort it is possible to significantly reduce the number of methodological, semantic, and implementation errors [8]. However, our experience indicates that the greater effort involved in formal validation of the resulting specification may not result in significant benefit.

The Nova tool shows the feasibility of the combined approach. We did benefit from our experience with DFT methods and from having developed Galileo. Someone trying to invent a modeling method or a POP-based tool from scratch would also face a long learning curve. What Nova does show is that, in less than two person-years and with the very modest effort of a graduate student, advisor, and domain expert, it is possible to build a tool with usability at least as good as 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 it will be as well-received as Galileo. Nova employs more aggressive use of the POP components, resulting in better responsiveness and more sophisticated behaviors and editing operations. Furthermore, Nova is the first tool which supports a formally defined and revised version of the DFT language.

Finally, the cost-effective development of a correct implementation of the analysis method remains a difficult problem which is beyond the scope of this research. The identification and possible development of suitable techniques for implementing the analysis engines of engineering tools is left for future work.

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

Recent work by Lédeczi et al. [22] describes a COTS-based generator for design environments. Given a meta-model of a modeling language, their tool generates a design environment. While their work is similar to our in terms of modeling and analysis environments and the technologies they employ, our work focuses on the integration of applications as components, whereas theirs deals primarily with the automatic generation of design environments. Furthermore the components that are used in the generation of the environments are mass-market applications such as Word and Visio.

As in our work, Succi et al. [29] 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

Several efforts are underway to formalize the notations used within software engineering, such as architectural diagramming notations [1], component connectors in software architecture [2], and the Unified Modeling Language [15]. These efforts place the notations used by software engineers on mathematically rigorous foundations. This provides a precise semantics for the notations which enables sophisticated analysis of models expressed in the languages. For example, architectural compatibility can be ensured by checking compatibility of connector types in a method analogous to type checking.

This work supports our basic argument regarding the necessity and applicability of formal methods to engineering models, in general. Our work investigates the use of formal methods in engineering domains other than software, and also seeks to establish an understanding of the benefits of formalization relative to the cost.

Our experience using Z/Eves is similar to that of Knight et al. [21], who used the PVS theorem prover on a modestly-sized nuclear power plant specification. They also found it difficult to formulate theorems and associated proof strategies, and were hindered by the poor usability of the toolset.

8.3. Tool development methods

Much work has been done on the design of tools for software engineering. More directly related to our work are techniques for building general modeling and analysis environments. Examples include the Generic Modeling Environment [22], MetaEdit+ [23], and DOME [17]. The approach in this research is to invest in the development of a reusable modeling framework which 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 generic 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, our work assumes that the modeling language has a semantics which can not be easily captured using the hierarchical and constraint-based semantics used by generic frameworks such as GME.

9. Conclusion

In this paper we presented and evaluated an approach to developing sound modeling methods and high quality

software tools to support them. Using packages as components enabled the development of industrially effective modeling tools at low cost. The enthusiastic acceptance of one such tool by NASA engineers is strong evidence that the result is acceptable. Using formal methods fundamentally improved the soundness of a novel, sophisticated, and industrially important modeling method. The Nova tool combines these two components, implementing a formally specified and validated language in an improved package-based interface. Nova demonstrates—for the first time, to the best of our knowledge—that it is possible to develop sound methods and to make them readily available for practical use at a cost that is modest by any industrially relevant measures. It took less than two person years to develop the specification for a new version of the modeling language, develop an initial implementation of the new language, and to embody them in a from-scratch package-based tool.

We believe this work has the potential to have a significant impact on engineering practice in two dimensions. First, by dramatically reducing the cost to deploy modeling methods via industrially effective software tools, it promises to catalyze the transition of such methods from laboratories into practice. Second, it has become clear that using modeling methods without sound semantic foundations in critical industrial, governmental, and military engineering design activities is fraught with risk. This paper shows that well known formal methods and a willingness to work with domain experts are enough to significantly promote the cost-effective formalization, correction, improvement, documentation, validation and implementation of sophisticated modeling methods. It thus appears reasonable to start to recognize and question the use of computerized modeling methods lacking adequate formal foundations for their semantics.

Acknowledgements

This work was supported in part by the National Science Foundation under grant ITR-0086003. We thank reliability engineers at NASA Langley Research Center, Johnson Space Center, and across the organization for agreeing to participate in our survey. We also warmly thank our expert collaborators in reliability engineering, especially Professor Joanne Bechta Dugan, without whom this work would not have been possible.

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 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.

[7] 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.

[8] 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.

[9] David Coppit, Kevin J. Sullivan, and Joanne Bechta Dugan. A Formal Semantics for Dynamic Fault Trees. *This document is available to reviewers on request on a confidential basis pending resolution of certain IP issues.*

[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] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. In *The Unified Modeling Language—Workshop UML'98: Beyond the Notation*, pages 336–48, Mulhouse, France, 3–4 June 1998.

[16] Les Hatton and Andy Roberts. How accurate is scientific software? *IEEE Transactions on Software Engineering*, 2(10):785-797, 1994.

[17] Honeywell. DOME users' guide. URL: http://www.htc.honeywell.com/dome/support.htm.

[18] Michael Jackson. Problems, methods and specialisation. *Software Engineering Journal*, 9(6):249-55, November 1994.

[19] Xiaoping Jia. *ZTC: A type checker for Z. Notation user's guide*. URL: `http://se.cs.depaul.edu/fm/ztc.html`.

[20] 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.

[21] J. C. Knight, Colleen L. DeJong, Matthew S. Gibble, and Luís G. Nakano. Why are formal methods not used more widely? Fourth NASA Formal Methods Workshop, Hampton, Virginia, September 1997.

[22] Akos Ledeczi, 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.

[23] MetaCase Consulting. Domain-specific modeling: 10 times faster than UML. URL: `http://www.metacase.com/papers/index.html`.

[24] 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.

[25] Mark Saaltink. The Z/EVES system. In *ZUM '97: Z Formal Specification Notation. 11th International Conference of Z Users. Proceedings*, pages 72–85, Berlin, Germany, 3–4 April 1997. Springer-Verlag.

[26] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 7(6):15-24, November 1990.

[27] J. M. Spivey. *The fuzz manual*. URL: http://spivey.oriel.ox.ac.uk/~mike/fuzz/

[28] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[29] Giancarlo Succi, Witold Pedrcyz, Eric Liu, and Jason Yip. Package-oriented software engineering: a generic architecture. IT Professional, 3(2):29-36, March-April 2001.

[30] 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.

[31] 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.

[32] 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.

[33] H. A. Watson and Bell Telephone Laboratories. Launch control safety study. Technical report, Bell Telephone Laboratories, Murray Hill, NJ, 1961.