Cost-Effective Tools Supporting Semantically Sound Domain-Specific Languages

David Coppit
University of Virginia Computer Science
david@coppit.org

Kevin J. Sullivan
University of Virginia Computer Science
sullivan@virginia.edu

Abstract

Domain-specific modeling languages and tools can deliver enormous computational leverage to domain experts. Such languages and tools are most valuable when they have a number of properties. First, language syntax and semantics must be based on abstract, formal specifications if models and results are to be trusted. Second, tools must implement these languages faithfully. Third, development costs must be low because they cannot be amortized in mass markets. Fourth, tools must be powerful in function and easy to use by domain experts. There are strong tensions among these requirements: formal methods, ease of use, powerful function, and faithful implementation conflict with low cost. We present an approach to overcoming this tension based on a pairing of two approaches. First, we apply formal methods, focused narrowly: not on the tool, but only on its language. Second, we use modern shrink-wrapped packages as vehicles for editing models in these languages. We hypothesize that this approach can help resolve the conflict we identified. Domain-specific languages (unlike tools that support them) are often small enough to be reasonable targets for formalization. Using packages as components reconciles cost, function and usability for the rest of the tool. To test this combination, we are applying it in the experimental design of a new dynamic fault tree tool similar to Galileo. The tool implements a new graphical language based on a formal specification of an abstract syntax and semantics of dynamic fault trees. It uses a domain-specific extension of Microsoft Visio for manipulating models. Overall the tool delivers a low-cost capability to edit models having abstractly and precisely specified meanings, unlike other tools in this category.

1. Introduction

An important role for software is to increase the productivity of non-programmer domain experts, enabling them to perform valuable new tasks. Engineers, for example, often use modeling and analysis tools to gain insights into properties of complex engineering designs. To use such a tool the engineer creates a model on the computer (often a drawing), from which the computer calculates performance estimates.

To be most valuable such tools and modeling notations have to have several properties. First, tools should have usability and functionality similar to mass-market packages. Second, if users are to trust their models and analysis results, the syntax and semantics of the notations must be precisely and abstractly specified. Third, the software that implements the semantic analyses must be created in a way that justifies trust in its dependability. Fourth, development costs must be low for such tools to be viable in small markets.

Unfortunately, delivering such capabilities is hard because there is a strong tension between demands for low costs and the other properties. Consequently, tools often cost too much to develop, lack the usability and functionality that people need–and so cost too much to use–and present users with seductive graphical notations for which no precise semantics have been defined (except in impenetrable computer codes).

We have developed an approach to relieving this tension based on the novel synthesis of two strategies. First, viewing the domain-specific modeling notation as a formal language, we use denotational semantics and formal methods techniques, focused narrowly, to create precise and abstract definitions of the language syntax and semantics. Second, we leverage programmable mass-market software to support manipulation of models–e.g., graphical drawings–expressed in the language.[1]

In earlier work we presented the Galileo tool, which showed that mass market packages can support manipulation of engineering models (namely for dynamic fault trees) [2, 8, 9]. We have also presented an abstract syntax and formal semantics for an improved dynamic fault tree language [3]. Here we make two new contributions. First, we present a new graphical syntax for dynamic fault tree

[1] This research is not intended to advance our ability to program analysis functions with high confidence given their specifications. That is a general software development problem that is beyond the scope of this work.

modeling based on our formally defined abstract syntax. Because that syntax already has a semantics (based on failure automata and Markov chains as semantic domains) the concrete graphical language inherits that semantics. Second, we present a graphical tool for this language based on the Visio drawing program. The result is a tool for modeling fault-tolerant systems which represents a step toward a more general tool for *trustworthy, affordable analysis of domain-specific models having well defined meanings*.

Section 2 describes our previous work formalizing and revising Dugan's dynamic fault tree framework. Section 3 then presents the derivation of our new graphical language from the revised formal specification of DFTs, and compares the new language to the original. Section 4 presents a prototype Visio-based tool for creating, editing, viewing, storing, and printing drawings in this new graphical language. The tool, built in just a few weeks, is a front end for a forthcoming analysis function based on our formal specification. Section 5 discusses some related work. Section 6 concludes.

2. Background: Formalization and Revision of Dynamic Fault Trees

In this section we present an overview of our previous work in which we formalized the DFT modeling language, and then revised the language based on issues discovered during the specification process. In the next section of this paper we show how such clarifications made using the abstract specification of the modeling framework resulted in a simpler and cleaner concrete language.

Traditional *static* fault trees [10] model how boolean combinations of component-level failure events produce system failures. For example, traditional constructs include the AND, OR, and KOFM gates, which model all, any, and minimal occurrences of events in a system. Dynamic fault trees [1, 4] (DFTs) extend traditional fault trees, allowing the engineer to model dynamic systems with sequences of failure events, pools of spare components, common-cause failures, and imperfect coverage of failures. For example, users can model the use of replacement components with the cold, warm and hot spare gates, the "temperature" of which indicates whether an unused spare does not fail, fails at a reduced rate, or fails at its normal rate. Another construct, the functional dependency (FDEP), models common-cause failures in the system where a trigger event causes immediate and simultaneous failure of the dependent events.

2.1 Formalization Of The DFT Specification

In previous work [3] we formalized the dynamic fault tree modeling language, providing, for the first time, a complete mathematical semantics for the framework. We structured the formal specification in a denotational style that separates two domains: an abstract syntax and an underlying semantic domain. The abstract syntax specifies the essential content of a given model in abstract mathematical terms. The semantic domain is a domain in which models are well understood and directly solvable: e.g., differential equations, Markov chains, binary decision diagrams, etc.

In this paper we will focus on the *concrete syntax*, which specifies the form of the models that the user manipulates directly with a tool. Each element of the concrete syntactic domain is associated with a corresponding element of the abstract syntactic domain. Then, using the specification, each of those is mapped to a corresponding element of the semantic domain. Thus, each concrete model that the user defines, sees and manipulates has an associated model in the semantic domain.

We formalized DFTs using the Z specification language [7]. Z is a language based on predicate logic and typed set theory, and supports specifications built using a structuring mechanism called the schema. A schema defines a type by specifying the state components of an element of the type, as well as invariant relations over these state components that are satisfied by all elements of the given type. State components of a schema can be basic types, given types, or other schemas. In addition, a *schema calculus* provides mechanisms for composing smaller specifications into larger ones.

The schema shown in Figure 1 defines the fault tree type. The portion of the schema above the middle horizontal line defines the state components, and the portion below the line defines the invariants over the state components. Although many of the details have been omitted, this example provides some understanding of the formalization of DFTs and introduces the predicates that define valid combinations of DFT constructs.

The three lines beginning on line 2.1 incorporate the definitions of the gates, invariants, and basic events. Their definitions, not presented here, establish important constraints on the range of values of parameters, the failure context in which the gates are defined, the connectivity, etc. The *events* set (line 2.2) is the set of all events in the fault tree, and *inputs* is partial function from events to sequences of events. (*inputs* is a partial function because basic events are events but do not have inputs.) At line 2.3 we declare a single event in the fault tree that is the system level event.

Some constraints on the fault tree elements are implied in the declarations. For example, the events state component is declared as a finite set, which places a constraint on the cardinality of events. In addition, there are several predicates over the state variables. For example, the predicate on line 2.4 states that event identifiers are unique. The three predicates starting on line 2.5 state that only gates can

$$
\begin{array}{l}
\textbf{FaultTree} \\
\hline
Gates \quad (2.1) \\
Invariants \\
BasicEvents \\
events : \mathbb{F}\, Event \quad (2.2) \\
inputs : Event \rightarrow\!\!\!\!\!\!+ \operatorname{iseq} Event \\
systemEvent : Event \quad (2.3) \\
\hline
\langle basicEvents, gates \rangle \text{ partition } events \\
\forall e_1, e_2 : events \bullet e_1.id = e_2.id \Leftrightarrow e_1 = e_2 \quad (2.4) \\
\operatorname{dom} inputs = gates \quad (2.5) \\
\forall g : gates \bullet \operatorname{ran}(inputs(g)) \subseteq events \\
\forall g : gates \bullet \neg\, IsInputTo(g, g, inputs) \\
systemEvent \in events \quad (2.6) \\
\neg\, (\exists g : gates \bullet IsInputTo(systemEvent, g, inputs)) \\
systemEvent.replication = 1 \\
\forall sg : spareGates \bullet \operatorname{ran}(inputs(sg)) \subseteq basicEvents \quad (2.7) \\
\forall sg : spareGates;\ be : basicEvents \mid IsDirectlyInputTo(be, sg, inputs) \bullet \\
\quad \neg\, (\exists g : gates \setminus spareGates \bullet IsDirectlyInputTo(be, g, inputs)) \\
\forall s : seqs \bullet \operatorname{ran} s \subseteq events \quad (2.8) \\
\operatorname{dom} fdeps \subseteq events \quad (2.9) \\
\forall d : \operatorname{ran} fdeps \bullet \operatorname{ran} d \subseteq basicEvents \\
\forall g : gates \bullet IsInputTo(g, systemEvent, inputs) \quad (2.10)
\end{array}
$$

Figure 1. Z Specification Of The Domain Of Abstract Fault Trees

have inputs, the inputs must be one of the other events in the fault tree, and no gate can be input to itself (directly or indirectly). These predicates define the abstract syntax of fault trees—the combinations of abstract elements which are considered to be legal fault trees.

2.2 Revision Of The DFT Specification

Formalization of the DFT modeling framework revealed important issues and subtle concepts in the specification. In light of these insights, we improved the framework by resolving or clarifying the issues in the specification. This section summarizes a few of the issues described in previous work [3], and the changes we made. These changes removed special cases and redundancy in the modeling framework at the level of the abstract representation, and had a direct impact on the resulting concrete representation.

Some revisions largely involved clarification of our understanding of the model. For example, the functional dependency and sequence enforcing constructs were previously treated as gates, even though they did not compute an output based on the value of their inputs. We renamed these constructs *constraints* because they constrain the sequences of events that can occur in the system. Similarly, the transfer gate construct allows the user to refer to another gate or basic event by name. However, it is not a gate because it has no inputs. We renamed the transfer gate an *indirect connector*, because, like a direct connector, it connects the output of one gate or basic event to the input of a gate or constraint. The difference is that the association is indirect—the indirect connector refers to the gate or basic event by its name.

Other revisions involved removing unnecessary and error-prone redundancy. For example, the KOFM gate's M value represents the total number of inputs to the gate, and is redundant because it can be computed from the inputs to the gate. Having to properly compute this value increases the burden on the modeler, and increases the chances of error. Our solution was to remove the M value, and rename the KOFM a *threshold* gate, where the threshold value serves the same purpose as the K value of the KOFM gate. Another obvious area of redundancy was the use of three types of spare gates. Our formalization effort revealed that the cause of the redundancy was lack of orthogonality in the language—sparing behavior and failure rate attenuation need not be tied to the same modeling construct. Our solution was to remove the attenuation semantics from the spare gate and represent it solely as a dormancy value for the basic event. This change allowed us to replace three types of spare gates with a single spare gate, and also allowed greater modeling flexibility than was allowed by the previous constructs.

Another type of revision was the removal of special cases. For example, the original formulation of the spare gate included a special *primary input*. This input ensured that every spare gate would have an operational component at the start of the system's mission time. Our specification effort made it clear that the primary input was not required. As a result, we decided to make the inputs of the spare gate uniform, and to properly handle the semantic issues associated with not having a primary input.

3. Deriving A New Graphical Language for Dynamic Fault Trees

In this section we describe how we derived a new version of the concrete language based on our formal specification.

3.1 Lexical Elements

The lexical elements of a language are the concrete representations of the language elements—the spelling of the words. In terms of a graphical language, the lexical elements are the shapes shown in a drawing.

Four concerns guided our design of the DFT shapes. The first was adherence to the formal specification. For example, if the formal specification stated that an event did not have an input value, then the corresponding shape should

not have an input connection point. The second concern was compatibility with legacy shapes—we did not want to force the user to re-learn shapes. The third concern was the intuitive nature of the language—any new shapes should be suggestive of their meaning. Finally, we did not want to design a language that would be difficult to implement. These four concerns were not always in agreement—sometimes we were forced to compromise one for another.

The state components of our specification indicated which portions of the language needed lexical representation. In addition to the basic event, our specification partitioned the set of gates into five types of gates (AND, OR, Threshold, PAND, and Spare) and two types of invariants (FDEP and SEQ). Our resulting DFT language has concrete lexical constructs for each of these eight abstract constructs. In addition, our specification abstracts the notion of connectors, representing inputs implicitly. In the graphical language, we included connectors to make input relations explicit: a direct connector which directly connects shapes, and an indirect connector which connects shape by naming the input shape.

We restricted the connections to shapes based on the specification (Figure 1). For example, the input constraints starting on line 2.5 allowed us to determine that all gates must have input connection points, and all events (basic events and gates) must have output connection points. In our specification a functional dependency has both a trigger input and a dependent event input, so the corresponding drawing shape has a connection point for both. Similarly, the sequence enforcer is a relation over a sequence of inputs, so it has an input connection point.

The resulting DFT lexical elements are shown in Figure 3. These shapes reflect the modifications made to the language in the abstract specification. For comparison, the original constructs are shown in Figure 2. The new lexical elements include a single spare gate, and it does not have a separate primary input. We retained several of the shapes, but also standardized the “label above smaller shape” structure of the SEQ and FDEP shapes, and added or modified the shapes of several gates in order to provide more intuitive representations. For example, the threshold gate has a “T” like structure, the spare gate has several sub-parts, and the sequence enforcer suggests an ordering. We also renamed the shape types based on the clarification in the specification of the differences between gates, basic events, constraints, and connectors.

In studying the specification, we realized that certain shapes had ancillary information essential to understanding the DFT. In particular, threshold gates have threshold values, and basic events have replication. We decided to represent these explicitly in the shapes. In addition, we needed a representation for the ordering of inputs to order-dependent gates, so we decided to indicate the order of an input as a number on the connector. Unlike threshold and replication, we decided not to explicitly represent basic event models in the fault tree drawing. See Section 4.2.1 for more information.

3.2 Grammatical Structure

The grammatical structure of a language describes the allowable combinations of lexical elements—the structurally correct sentences. In a graphical representation, the grammatical structure indicates the legal connections of shapes and values for the attributes of the shapes.

Many of the structural rules imposed by the formal specification are implicit in the drawing. For example, rules that state that the number of basic events is finite, or that the system level event must be in the set of events for the fault tree are implicitly fulfilled by the definition of the concrete graphical representation. Similarly, the constraint that a basic event can not have an input is implicit in the fact that we did not give the basic event an input connection point.

The other structural rules of a graphical fault tree correspond to the abstract structural rules described in Section 2.1. For example, the fifth predicate in Figure 1 disallows cycles in the fault tree, and is implemented as a function that checks for cycles in the fault tree drawing. We chose to allow the user to violate most of the global structural constraints while constructing the drawing, invoking a “validity check” which ensures that all the constraints are satisfied. See Section 4.3 for more information.

In a few cases, the usability of the language outweighed our desire to adhere to the formal specification. For example, the replication for gates is not represented because it is always 1 in the abstract specification. Similarly, we only require the ordering of inputs to be specified for gates whose semantics are order-sensitive. Lastly, although the specification defines the semantics of gates for no inputs, we require at least one input in the graphical language in order to avoid confusion on the part of the user.

4. A Dynamic Fault Tree Editor Built Using POP

Our implementation of the new graphical language was built using a volume-priced mass-market application—Microsoft’s Visio drawing tool. The construction of this modeling tool is part of an ongoing evaluation of the package-oriented programming (POP) [2, 9, 8] style of component-based software development. In this style, multiple mass-market applications are specialized and integrated to provide the bulk of the software system’s functionality at greatly reduced cost. Not only does the developer reuse the person-years of development effort typically em-

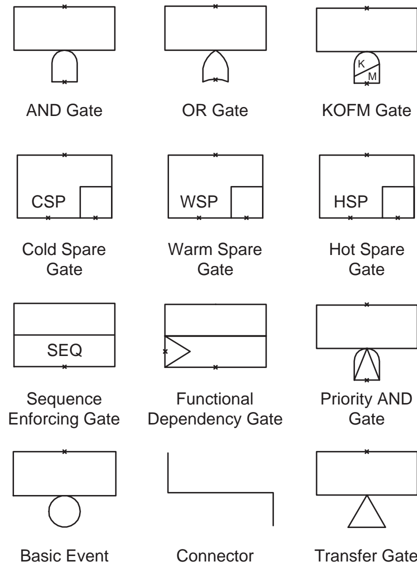

Figure 2. Original Depictions of DFT Shapes

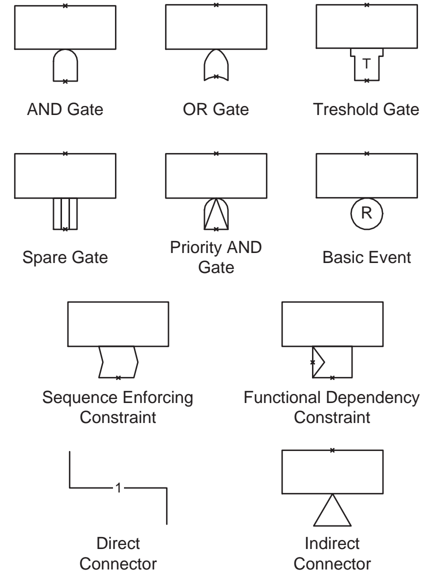

Figure 3. Revised Depictions of DFT Shapes

bodied in such packages, but the user enjoys mass-market pricing, high usability, and low training costs.

In this section we present our work specializing the Visio application for the modeling of dynamic fault trees. This work provides an additional data point demonstrating the feasibility of POP. In particular, it demonstrates that mass-market packages can support more aggressive specialization of behavior than previously shown.

4.1 Specialization Capabilities Of Visio

Applications such as those in Microsoft's Office suite provide the programmer certain specialization capabilities. In addition to user-visible mechanisms such as custom menus, applications also expose a programmer-visible object model. This object model allows the programmer to set properties of objects in the application, call methods on objects, or handle events raised by objects.

Visio provides user-level customization through the construction of custom *stencils* of drawing shapes. Stencils contain a *shapesheet* which is a spreadsheet with instructions for drawing the shape, as well as properties for its formatting, text label, grouping behavior, etc. For example, a shape may be locked against being flipped horizontally, or may contain context menu items which invoke software functions when selected.

Functions are implemented using a general-purpose programming language such as Visual Basic or C++. In addition to the arguments supplied to a callback, the programmer has access to any objects that are visible in the context of the function, such as the global `Application`, `ActiveDocument`, and `ActiveWindow` objects. From these objects, the programmer can access collections of objects related to various drawing elements.

Visio objects expose events allow behaviors to be intercepted and modified. There are drawing-level events associated with the opening of documents, the deletion shapes, and the connection of shapes. However, with the exception of the double-click of a shape, Visio does not expose low-level events associated with mouse movements or individual key presses.

4.2 Overview Of The Tool

Figure 4 is a screenshot of the tool. Visio provides much of the overall functionality, such as zooming, scrolling, formatting, saving, printing, etc. In addition, we specialized the interface in several dimensions, creating a complete, stand-alone editor for dynamic fault trees. First, a stencil of shapes has been added which contains the graphical depictions of the shapes as described earlier. 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. Third, certain functions which are inappropriate for DFTs have been removed or replaced in the interface. Fourth, the behavior of Visio has been enhanced—for example, Visio automatically checks for duplicate names

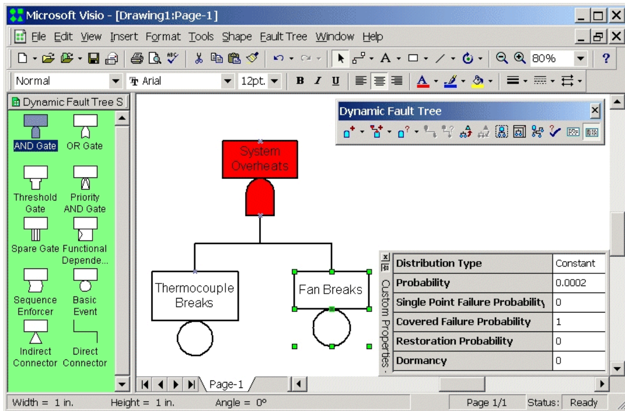

Figure 4. A Screenshot of the Tool

when a shape is given a name. Each of these is described in more detail in the following sections.

4.2.1 The Stencil

The stencil of shapes that we created implements the graphical depictions shown in Figure 3. In addition, the shapes in the stencil implement dynamic constraints. For example, the text box associated with a shape can be manually resized, and also automatically expands to accommodate long text. There is a "control point" that the user can also use to change the size of the DFT shape under the text box. In addition, each shape has a context menu for operations that can be performed on that shape.

The threshold gate and basic event also have an extra text field on the shape in which the user can enter the threshold value or replication. Because Visio only allows a shape to have one text box, we were forced to design the basic event and threshold shapes as a group of two shapes—one having the main text box, and one having the threshold or replication text box. This design complicated the shapes because a change to the size or position of one member of the group required that the other be updated to match. Unfortunately, we found the shapesheet model unable to easily handle the cyclic updates that result from the modification of a group member. As a result, we had to implement our update routines as a Visual Basic callback that would be invoked whenever a group member's position or size changed.

We integrated database support into the tool in order to allow the user to include instances of predefined basic events in DFTs. A context menu item allows the user to set the basic event's characteristics to those of an event selected from a database. In other words, one can store a generic "wheel assembly" basic event in the database, and import its characteristics into a "left wheel assembly" basic event and a "right wheel assembly" basic event. One can also easily add a new basic event to the database, or refresh an event's characteristics from the database.

4.2.2 Editing Operations

Domain-specific editors are often *structure editors* which maintain the validity of the model at all times by forcing the user to use validity-preserving operations for constructing the model. In contrast, Visio is a free-form editor that allows drawings to be constructed using generic low-level operations. Instead of attempting to change this design decision in the Visio application, we decided to implement a free-form fault tree editor, and augment this interface with validity-preserving, structure-based editing operations. Because the user may construct an invalid fault tree using the low-level editing operations, we also provide a validity check operation which can be used to verify that a fault tree is valid.

Each of the structure-based operations is implemented in terms of basic Visio objects, which provide for low-level, generic manipulation of graphical drawings. An earlier tool we built using Visio followed a bottom-up implementation which addressed the large degree of uncertainty in the functionality that can be efficiently implemented in terms of Visio objects. We followed the same approach, building a library of drawing manipulation functions appropriate for DFT manipulation. The high-level functionality was then implemented in terms of this library. For example, the "send subtree to page" operation is implemented in terms of a lower-level "select subtree" DFT operation, which is in turn implemented in terms of Visio's native shape and connection object functionality. Examples of the structure-based editing operations we implemented include "add shape and connect to selected shapes", "change shape type", "send subtree to page", "fit page to drawing", and "lay out selection or page".

4.3 Implementing Structural Constraints

The structural constraints of the formal specification can be divided into *local* and *global* constraints. Local constraints are those that are expressed in terms of one or a few constructs of the fault tree. Global constraints are those expressed in terms of all of the constructs of the fault tree. For example, the constraints which states that the trigger of an FDEP must have a replication of 1 is a local constraint, while the constraint which says that event identifiers must be unique is a global constraint.

We perform validation of local constraints as the fault tree is constructed, but defer most global constraints to the validity check. This decision also helps to address performance issues in Visio, because global checks typically require traversal of the entire drawing one more more times.

Performing such analyses as the fault tree is constructed would make the editor seems sluggish.

4.3.1 Inappropriate Functionality Modified Or Hidden

In addition to binding functions to shape behavior, the programmer can also use the Visio `Document` and `Application` objects to extend the interface by creating new menus and toolbars. Similarly, existing functionality can be removed, or rebound to perform a different function. For example, Visio provides two methods for viewing the custom properties of shapes. One method reacts dynamically to changes to the underlying shapesheet, while the other does not. Or solution was to rebind the static method to invoke the dynamic method instead.

4.3.2 Enhanced Behavior

An important and necessary specialization capability is the specialization of package behavior. In several cases we found the default behavior of Visio to be inappropriate or insufficient for our application. Using Visio's event interception mechanism we were able to redefine or augment the application's behavior in a number of important ways:

- **Duplicate names and invalid connections**: As shape names are changed, we check for duplicates and issue a warning if any are found. Similarly, when a connection is made, we break the connection if the connection between shapes is input-to-input or output-to-output.

- **Directed vs. undirected connectors**: Visio by default uses directed connectors. (They are directed even if they have no arrowheads.) In contrast, the connectors in the DFT language are not directed. Generally this is not a problem, but we discovered that the layout algorithm depends on the directivity of the inputs. As a result, when a connector is attached to a connection point on a shape, we intercept the connection event and flip the connector if necessary.

- **Proper hyperlink management**: When a page is deleted in Visio, hyperlinks between shapes become invalid because the page numbers change. As a result, we had to intercept the page deletion event and update the hyperlinks ourselves.

- **Automatic shape coloring**: We wanted the system-level event for the fault tree to be automatically colored red. To implement this, we had to intercept every connection event and shape name change in order to correctly color the connected shape.

- **Dynamically glued connectors**: Visio supports a feature called "dynamic glue" in which shapes are connected by a connector which automatically selects connection points on the shapes in order to minimize connector length. Unfortunately, this behavior is inappropriate for DFTs, where connection points are not interchangeable. Because Visio provided no user-level ability to disable this feature, we had to automatically detect dynamically glued connectors when a connection was established and convert the dynamic glue to static glue.

- **Custom properties window**: By default, Visio displays a custom properties window even if a shape has no custom properties. The only shape in the DFT framework which had such properties was the basic event, so we wanted to modify the behavior of the window such that it would only display only if a basic event is selected. In order to do this, we rebound the custom properties functionality to our own callback function which only shows the window if a basic event is selected.

4.4 Discussion

Our use of formal specification for the abstract representation of the modeling framework revealed opportunities to revise the graphical language. The resulting graphical language is simpler and more consistent. The implementation of the language was greatly informed by the specification, although some compromises were made for ease-of-use. Importantly, having the formal specification of the modeling framework allowed us to make conscious and well-informed tradeoffs between the abstraction of the specification and the ease of use of the language.

Our use of package-oriented programming allowed us to create a complete, richly functional modeling tool for dynamic fault trees in a fraction of the time that a from-scratch development effort would have required. Using the specialization capabilities of Visio, we were able to successfully modify the interface, functionality, and behavior to suit our needs.

The implementation of the modeling tool required about 4 weeks of effort by a single programmer, which includes about 1 week learning Visual Basic for Applications and the Visio object model. In addition to the stencil, the implementation consists of about 3800 non-comment, non-whitespace lines of code. The use of Visio provides a tremendous amount of reuse—provided that the user has the application, the components required to specialize it can be distributed in a 500 kilobyte package.

On the down-side, our current prototype implementation has poor evolution characteristics. If a document is created

from the DFT template and stencil, its associated implementation is bound to the document file, making upgrade difficult. However, it is possible to create a level of indirection, storing the implementation in a dynamically linked library. This allows the implementation to be upgraded independently from the document.

We found the specialization capabilities of Visio to be adequate for our needs. In particular, we found that the event-based mechanisms allowed for easy composition of behaviors. For example, we implemented an event handler to respond to the addition or deletion of shapes, automatically adding or removing hyperlinks between indirect connectors and the shapes to which they refer. Later, when we implemented the "send subtree to page" functionality, hyperlinks were automatically updated for shapes in the subtree because the event handlers where implicitly invoked as each shape when the subtree was deleted from one page and added to another.

We did encounter performance issues that forced us to modify our design slightly. For example, we discovered that determining the type of a shape was a very common operation that would access the stencil repeatedly. For this situation we amortized the cost by caching the objects retrieved from the stencil. We also found that we had to disable the on-the-fly checks performed in the event handlers while we were making changes to the document programmatically.

We also found Visio's `UpdateUI` application method to be particularly slow, which resulted in noticeable delays in the selection of shapes on the drawing page. As a result, we had to modify our design so that user interface updates are performed during idle times. In general, performance limitations can seriously compromise a design because the black-box nature of application packages does not provide the programmer any recourse for addressing poor package performance. In this case, our redesign resulted in acceptable performance, and discussions with Visio developers revealed that the upcoming Visio 2002 has a new user interface object model which has better performance properties.

5. Related Work

Our earlier work on the formal specification of dynamic fault trees [3] established, for the first time, a complete formal semantics of dynamic fault trees in terms of Markov chains. The representation that we chose to formalize was an abstract one—we abstracted away the details of connectors, graphical depictions of the shapes, etc. This paper addresses the design of a graphical language based on the specification, and the corresponding implementation of that language in a modeling tool.

The DIFTree tool by Dugan et. al [5] included a graphical DFT editor built using Tcl/Tk. This editor implemented the original DFT language, and was not based on a formal semantics. Although it lacked much of the functionality of the tool presented in this paper, both the graphical depictions of the shapes and structure-based editing operations were adopted in the present tool. In comparison to our 4 weeks of implementation effort, the DIFTree implementation took about 6 person-months.

This paper presents work that is part of an ongoing evaluation of package-oriented programming [2, 9, 8]. In earlier papers, we described the use of multiple packages integrated together in the Galileo tool for dynamic fault tree modeling and analysis. Our preliminary evaluation of the POP approach indicated that it has the potential to succeed, and that the model has the potential for both significant benefits and significant risks. In contrast to our previous work, our current efforts are driven by the use of a formal specification of the modeling framework. In terms of the evaluation of POP, the work presented here provides an additional data point for the feasibility of the model, demonstrating that designers can successfully implement more aggressive specialization of application behaviors.

Goldman and Balzer report on the use of Powerpoint as a platform for building a software architecture modeling and analysis tool [6]. They cite many of the same benefits that we discussed here and in earlier work. Our experience was different from theirs in that Goldman and Balzer found that they had to intercept low-level events not exposed by the application. In contrast, we found that all of our functionality could be implemented in terms of the object model exposed by the application, which allowed us to avoid understanding and utilizing the low-level Windows event model.

6. Conclusion

The contribution of this paper is a proof-of-concept evaluation of one part of this approach: the use of shrink-wrapped packages to present engineering models whose semantics are formally defined. We found the approach to be workable and cost-effective, developing an editor for a domain-specific modeling language having a formal semantics.

This tool is the front-end for the system we are building to evaluate our overall approach. Having completed the modeling capability, we must now turn our attention to the automated analysis of DFTs. We intend to use the formal specification in conjunction with a disciplined process to implement both an abstract software representation for dynamic fault trees and an engine for mapping them to Markov chains and then to analysis results.

Acknowledgements

This work was supported by the National Science Foundation, grant number ITR-0086003. Joanne Bechta Dugan

has helped us enormously over the last few years to understand dynamic fault tree analysis. We also acknowledge the efforts of Jake Cockrell and Miro Kresonja in developing a Visio-based implementation of the graphical interface for the original DFT language.

References

[1] M. A. Boyd. *Dynamic Fault Tree Models: Techniques for Analysis of Advanced Fault Tolerant Computer Systems*. PhD thesis, Duke University, Department of Computer Science, Apr. 1991.

[2] D. Coppit and K. 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.

[3] D. Coppit, K. J. Sullivan, and J. B. 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 Oct. 2000. IEEE.

[4] J. B. Dugan, S. Bavuso, and M. Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability*, 41(3):363–77, Sept. 1992.

[5] J. B. Dugan, B. Venkataraman, and R. 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 Jan. 1997.

[6] N. M. Goldman and R. M. Balzer. The ISI visual design editor generator. In *1999 IEEE Symposium on Visual Languages (VL'99)*, pages 20–27, Tokyo, Japan, Sept. 1999. IEEE.

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

[8] K. Sullivan and J. Knight. Experience assessing an architectural approach to large-scale systematic reuse. In *Proceedings of the 18th International Conference on Software Engineering*, pages 220–229, Berlin, Germany, 25–30 Mar. 1996. IEEE.

[9] K. J. Sullivan, J. C. Knight, J. Cockrell, and S. Zhang. Product development with massive components. In *Proceedings of the 21st Annual Software Engineering Workshop*, Greenbelt, MD, 4–5 Dec. 1996. IEEE.

[10] 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.