

Galileo: A Tool Built From Mass-Market Applications

David Coppit

Dept. of Computer Science
Thornton Hall
University of Virginia
Charlottesville, VA 22901 USA
+1 804 982 2291
david@coppit.org

Kevin J. Sullivan

Dept. of Computer Science
Thornton Hall
University of Virginia
Charlottesville, VA 22901 USA
+1 804 982 2206
sullivan@cs.virginia.edu

ABSTRACT

We present Galileo, an innovative engineering modeling and analysis tool built using an approach we call *package-oriented programming* (POP). Galileo represents an ongoing evaluation of the POP approach, where multiple large, architecturally coherent components are tightly integrated in an overall software system. Galileo utilizes Microsoft Word, Internet Explorer, and Visio to provide a low cost, richly functional fault tree modeling superstructure. Based on the success of previous prototypes of the tool, we are now building a version for industrial use under an agreement with NASA Langley Research Center.

Keywords

package-oriented programming, COTS, large component integration

1 MOTIVATION: FROM-SCRATCH DEVELOPMENT IN SMALL MARKETS

As the demand software continues to increase, it becomes apparent that line-by-line software development is too costly, resulting in software that, if produced at all, is lacking in the sophistication that users demand. This problem is most apparent in small markets, where the cost of developing software can not be easily amortized across a large number of buyers [5].

One such market is engineering modeling and analysis tools. As with most software, engineers expect modeling tools to have high usability, to have large feature sets, and to interoperate within the overall engineering process. In addition, the modeling frameworks can be complex and subtle, which increases the software developer's cost of correctly specifying and implementing the system. The tools must also have a high level of assurance in the models

and associated analyses because the tools are critical in that their results are used to make design decisions which, if incorrect, can have significant consequences. [16]

Consider the two screenshots in Figure 1. On the left is Reliasoft's BlockSim tool [14], which is used to model and analyze block diagrams. On the right is Microsoft's Visio [13], a general purpose drawing tool. Note the large amount of functionality they have in common: the ability to format shape labels, position shapes on a drawing page, scrolling, zooming, shape "stencils" from which to build diagrams, etc. The key point is that Visio's functionality closely matches the graphical modeling needs of such tools. The ability to specialize and integrate large application such as Visio promises to significantly lower the cost of developing tools.

2 PACKAGE-ORIENTED PROGRAMMING

Package-Oriented Programming is an architectural style in which large, architecturally coherent components are specialized and integrated as part of the overall system [16,17,18,19]. Large components can provide larger fractions of the overall functionality of the system, and can potentially lower integration costs because there are fewer components to integrate.

Large components of particular interest are applications such as the Microsoft Word document editor and the Visio graphical drawing tool. Such standard software packages deliver large amounts of functionality at a very low cost due to their mass-market nature. In addition, because such components are also popular stand-alone applications, users are often already familiar with them, and much of the application documentation applies when the application is used as a component [17].

There are several important questions that must be answered for such an approach to succeed. First, the extent to which component integration standards can address the problem of architectural mismatch [10] is not clear. Second, large components bind numerous design decisions [2], which means that they must provide suitable degrees of specialization that allow developers to utilize them effectively. Another problem is the effect of component upgrades on the operation of the system. Unlike traditional

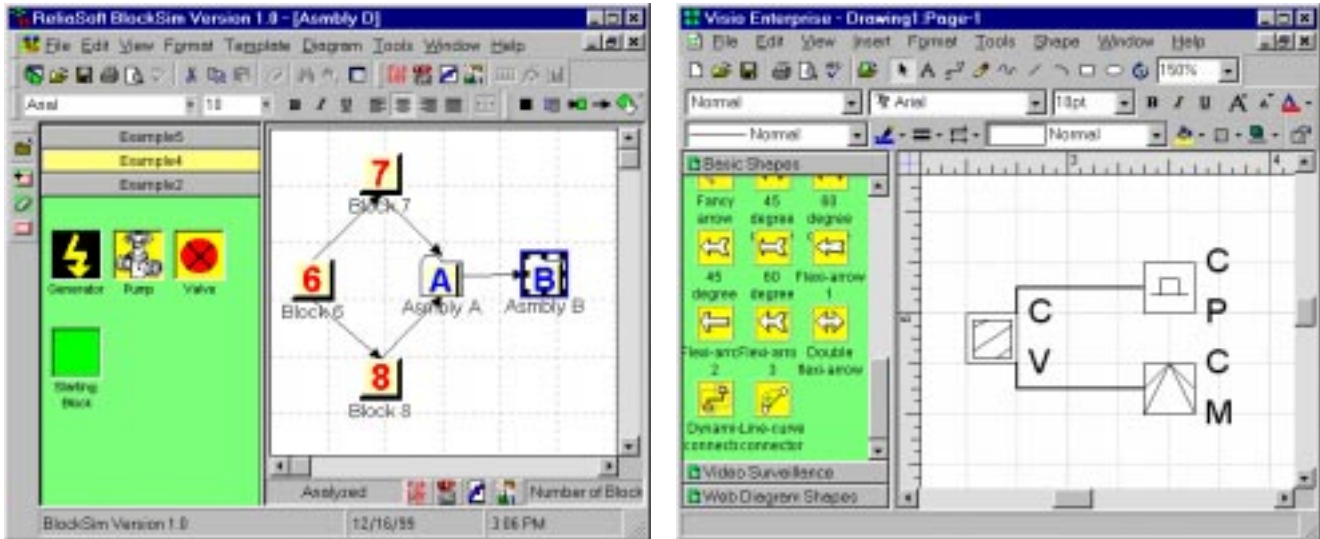


Figure 1: Comparison of BlockSim and Visio

components, new versions of mass-market applications may be acquired independently of the developer, and it is not clear the degree to which new components will be backwards compatible. Other issues include the large and undocumented limitations, behaviors, and interfaces of the components, the problem of testing such components, integrating their user interfaces, etc.

3 GALILEO FEATURES

Galileo is a dynamic fault tree analysis tool used by engineers to model the reliability of a system. Fault trees [20] are models of system failure, where particular combinations and orders can cause overall system failure. Galileo hosts Dugan's *DIFTree* analysis methodology [8,11], which is capable of solving fault trees quickly using an innovative divide-and-conquer methodology. The *DIFTree* approach first modularizes a fault tree into statistically independent subtrees, which are then solved using either a dynamic subtree solver or a static subtree solver.

Galileo's features include:

- the ability for the user to edit a fault tree in either a textual or graphical representation
- automatic rendering from the textual view to the graphical view, or vice-versa
- exploitation of the user interfaces of off-the-shelf packages, e.g., zoom, find-and-replace, print preview, etc.
- exploitation of the user's familiarity with common applications, significantly reducing training costs
- integration of the separate interfaces of the component applications into a single Galileo interface
- on-line documentation through an embedded internet

browser and World-Wide Web pages

- support for fault tree specific editing operations

A complete description of the current public version of Galileo has been published elsewhere [17]. Here we present the highlights and most recent developments.

1 Careful Reimplementation of the DIFTree Core

Early versions of Galileo used a legacy implementation of the core *DIFTree* engine that was carefully extracted from the existing Unix-based tool and then embedded in Galileo [6]. Because of serious maintainability and evolvability issues, the engine is being redesigned and reimplemented from the ground up.

The current developmental version of Galileo uses a more flexible modularizer. By identifying and clarifying the distinction between *subtrees* and *modules*, we were able to develop an algorithm that first divides a fault tree into subtrees, and then combines them according to one of a number of possible policies. For example, Dugan's statistically independent modularization can be replaced with Anand & Somani's modularization policy, which trades performance over accuracy [1].

Galileo also has a new dynamic fault tree solver based on a carefully written semi-formal specification of the fault tree to Markov chain translation algorithm [12]. This new implementation repairs several bugs in the previous one, and promises to be more maintainable. An analogous reimplementation of the static solver is currently underway.

2 New User Interface

Figure 2 shows a screenshot of the current developmental version of Galileo. The upper right sub-window contains the graphical representation of a simple fault tree, and the lower sub-window contains the corresponding textual representation. The upper left sub-window is the stencil

both of Microsoft Word (95 or 97) and Visio Technical or Enterprise (4.1 to 5.0). If Internet Explorer is available, Galileo will also use it to display hypertext-based documentation. A minimum computer is a 166 MHz "Pentium-class" computer, with 32 MB of main memory and 50 MB of disk space. Installation involves the unzipping of the archive, and the execution of a standard setup program.

ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grants CCR-9502029 (CAREER), CCR-9506779, CCR-9804078, and CCR-9804078, MIP 95-28258, and by NASA Langley Research Center under contract NAS-99098.

REFERENCES

1. Anju Anand and Arun K. Somani, "Hierarchical analysis of fault trees with dependencies, using decomposition," In Proceedings of the Reliability and Maintainability Symposium, January 1998, pages 69-75.
2. Ted Biggerstaff and Charles Richter. Reusability framework, assessment, and directions. IEEE Software, 4(2):41-9, March 1987.
3. Mark A. Boyd, Dynamic Fault tree models: Techniques for Analysis of Advanced Fault Tolerant Computer Systems, Ph.D. thesis, Department of Computer Science, Duke University, 1990.
4. David Coppit and Kevin J. Sullivan. Formal specification in collaborative design of critical software tools. In *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium*, pages 13-20, Washington, D.C., 13-14 November 1998. IEEE.
5. David Coppit and K. J. Sullivan, "Multiple mass-market applications as components," Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, 4-11 June 2000.
6. David Coppit and Kevin J. Sullivan. "Unwrapping." Technical Report CS-98-08, Department of Computer Science, University of Virginia, 4 May 1998.
7. Joanne Bechta Dugan, Salvatore J. Bavuso and Mark A. Boyd, "Dynamic fault tree models for fault tolerant computer systems," IEEE Transactions on Reliability, Volume 41, Number 3, pages 363-377, September 1992.
8. Dugan, Venkataraman, and Gulati, "DIFtree: A software package for the analysis of dynamic fault tree models," Proceedings of the 1997 Reliability and Maintainability Symposium, January 1997.
9. 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 (to appear).
10. David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. IEEE Software, 12(6):17-26, November 1995.
11. Rohit Gulati and Joanne Bechta Dugan, "A modular approach for analyzing static and dynamic fault trees," in Proceedings of the Reliability and Maintainability Symposium, January 1997.
12. Ragavan Manian, "Software architectural design and implementation of a dynamic fault tree analyzer," Master's Thesis, University of Virginia, August 1998.
13. Microsoft Corporation, Visio. <http://www.visio.com/>
14. Reliasoft. <http://www.reliasoft.com/>
15. J.M. Spivey. The Z Notation: A Reference Manual. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
16. Kevin J. Sullivan, Jake Cockrell, Shengtong Zhang, and David Coppit, "Package-oriented programming of engineering tools," In Proceedings of the 19th International Conference on Software Engineering, pages 616-617, Boston, Massachusetts, 17-23 May 1997, IEEE.
17. Kevin J. Sullivan, Joanne Bechta Dugan and David Coppit, "The Galileo Fault Tree Analysis Tool," Proceedings of the 29th International Conference on Fault-Tolerant Computing (FTCS-29), 1999.
18. K. J. Sullivan and J.C. Knight, "Building Programs from Massive Components," in Proceedings of the 21st Annual Software Engineering Workshop, Greenbelt, MD, Dec. 4-5, 1996.
19. 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.
20. United States Nuclear Regulatory Commission, Fault Tree Handbook, NUREG-0492, 1981.

