

Multiple Mass-Market Applications as Components

David Coppit

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

Kevin J. Sullivan

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

ABSTRACT

Truly successful models for component-based software development continue to prove elusive. One of the few is the use of operating system, database and similar programs in many systems. We address three related problems in this paper. First, we lack needed models. Second, we do not know the conditions under which such models can succeed. In particular, it is unclear whether the notable success with operating systems can be replicated. Third, we do not know whether certain specific models can succeed. We are addressing these problems by evaluating a particular model that shares important characteristics with the successful operating system example: using compatible PC packages as components. Our approach to evaluating such a model is to engage in a case study that aims to build an industrially successful system representative of an important class of systems. We report on our use of the model to develop a computational tool for reliability engineering. We draw two conclusions. First, this kind of model has the potential to succeed. Second, even today, the model can produce significant returns, but it clearly carries considerable risks.

Keywords

component-based software, package-oriented programming

1 INTRODUCTION

Because software is far less costly per copy when many copies can be sold than it is to build initially, researchers have tried for many years to devise models for component-based software development (CBSD) [24,25]. Components are independently developed software modules, and CBSD the construction of systems largely through the integration of such components. CBSD research has led to a deeper understanding of the need for components to have explicit

context dependencies, clear specifications, conformance to standards, intellectual property protection, etc.

Many researchers express optimism that component-based software development will succeed where previous reuse approaches have largely failed to meet expectations [3,8,25]. However, many are also increasingly circumspect or even skeptical, citing a number of technical and non-technical barriers to success. Designing software for reuse is relatively expensive. Conflicting architectural and other assumptions are a problem [11]. Integrators do not control component design. Understanding components can be costly [14]. Incomplete knowledge of the properties of large components is typical and perhaps even inevitable [5]. Even if existing components can be made to work, there is no guarantee that future versions will work. Global analysis is hard when multiple components are used in a system. This problem is exacerbated when components are integrated dynamically [24]. We lack demonstrably effective payment schemes for many CBSD models.

In fact, there are relatively few successful CBSD models. Traditional math, user interface, data structure, and I/O libraries are one example. The Unix pipe and filter model is another. Lampson observed that one of the most successful models is that of operating systems, databases and web browsers. These massive components provide functions for a wide variety of systems. He has also expressed the opinion that this approach is unlikely to generalize, owing to problems of cost, mismatch, and complexity [14].

Prospects for broader success with CBSD thus remain murky. In this paper, we address three interrelated problems. First, we lack needed CBSD models. Second, we do not adequately understand the conditions under which such models might succeed. Third, we do not know whether certain promising models that have not yet been studied carefully do have the potential to succeed.

Repeated experience has shown that just speculating on the likelihood of success of a CBSD model is untrustworthy. On the other hand, the issues are too complex to be treated analytically. We believe that a rigorous evaluation of a relatively untested CBSD model requires that it be tested

against the demands of real applications in practice. Ideally, a model would be applied to many systems, but that is not a feasible experimental method. Our approach to addressing these problems is thus to apply such a model in a case study that aims to produce an industrially viable system representative of a class of important applications.

We emphasize industrial viability in order to ensure that the demands of real users apply in the experimental context. It is always possible to build a toy system. The question is, can a model contribute significantly to the production of real systems. We emphasize the design of a systems that is representative of an important class of systems to help ensure that results of a single-point case study generalize.

In this paper we report on our ongoing evaluation using this approach of one promising CBSD model: the use of mass-market applications as components, which we call *package-oriented programming* (POP). In Section 2 we discuss this CBSD model. In Section 3 we describe our evaluation approach. In Section 4 we describe the experimental system that we are developing and our reasons for selecting it as a case study. In Section 5 we present data from our system building experience. In Section 6 we present our evaluation of the POP approach to date. In Section 7 we discuss related work. Then in Section 8 we conclude.

2 THE PACKAGES-AS-COMPONENTS MODEL

The CBSD model that we are evaluating is based on the use of multiple, architecturally compatible, mass-market packages as large components. We call this model Package-Oriented Programming (POP). In this paper, we examine the instance of this model with packages such as those in the Microsoft Office suite and Visio Technical as components. Such packages provide not only user interfaces but also developer interfaces that allow them to be customized and to be driven by programs.

The idea is to use such packages to provide most of the functions of a given system in dimensions that the packages address: textual editing, graphical drawing, hypertext browsing, etc. Such packaging is often the bulk of the system. Avoiding the need to build it can lead to order-of-magnitude cost reductions.

There are two distinct CBSD models that use packages as components. In one, a single component is employed as a *platform* upon which to build a system. Using a database management system as a platform is an example that has been employed successfully for years. In work related to ours, Goldman and Balzer have used PowerPoint as a platform for software architectural drawings and animations [12]. Modern package vendors continue to favor this model, perhaps for economic reasons. Visio presents an API, for example, but it is presented as a platform upon which vendors can build applications.

The second model is one in which *multiple* component applications are integrated tightly in a single application. We refer to this multi-package model by the term POP, and it is this model that we address in this paper. We believe that it is an important model because many systems need functions in multiple orthogonal sub-domains, each of which tends to be addressed by a different package. For example, our experimental tool, Galileo, requires technical drawing, text editing, and hypertext browsing functions.

This model is not a new idea. Brooks [3] and others have speculated that it has considerable promise. There are good reasons to evaluate the POP model. If it works, the advantages can be enormous [12,22].

- The components provide tremendous function to system integrators within general sub-domains that are important across broad families of systems, e.g., textual and graphical editing, database, etc.
- The price to obtain the given functions is extremely low, despite high development costs, because packages are sold not only as components but, by the millions, as end-user applications.
- System design costs are reduced significantly because the components provide so much functionality that only a few need to be integrated.
- The cost to learn and use systems built from such components is reduced because people already know the particular components and the style of components.
- The cost of components to end-users is amortized over many uses, including use as stand-alone applications. Many enterprises already own such packages.
- The problems of component licensing and payment are avoided because end-users buy the components. The integrator just sells application-specific code, including “glue code,” which invokes the installed packages.
- Modern application suites impose integration standards that mitigate problems of architectural mismatch [23].
- Applications are upgraded continually with powerful new features, providing system integrators with large increments of function at very low cost. Evolution is a two-edged sword, in that new versions might not work. However, large installed bases often drive component vendors to maintain backward compatibility.
- Component vendors can be influenced, in some cases, by component integrators to provide features and fixes, provided the vendor is convinced that there is a valid business case for doing so.
- Documentation effort is reduced because the functions provided by the components are already documented.

Despite the obvious attractions of the model, it is not clear that it will work with today’s components. To the best of

our knowledge, outside of our work, there have been no serious evaluations of the model published in the literature. There are many reasons for sobriety.

- Large components bind design decisions that can make it hard or impossible to meet requirements [1,17].
- Even if a set of components is matched to current requirements, it might be hard or impossible to meet new or changed requirements using given components.
- The size of modern components makes complete and accurate specification of their interfaces unlikely. In the resulting uncertainty, component inadequacies can go unrecognized until late in development, and given components can be used incorrectly or sub-optimally.
- Even if a component is specified fully, the complexity of the specification can make it hard to use effectively.
- Large components consume resources that might not be used in a given application. In practice, this might or might not be a problem for a given user, given the rapidly decreasing cost of hardware.
- A component application programming interface might not be consistent with its user interface. For example, a component might not make all user-level functions available to the integrator. Such a mismatch can mislead the integrator into believing that a component can perform functions that actually are not available, resulting in nasty surprises late in development.
- Components provide their own user interfaces. This removes complexity from user interface development, but creates the problem of specializing and integrating the interfaces and keeping users from invoking functions that should not be used in the system context.
- Because POP components run in their own address spaces, integrating them incurs the cross-application communication penalty. In addition, we have found that components can display internal inefficiencies when manipulated via their APIs.
- Components are distributed in binary form, which helps protect vendor's intellectual property. However, without source code it is impossible to investigate or repair limitations, or to adapt components in ways not explicitly provided by the vendor. Of course, even if source code for old, large applications were available, it is unclear that it would be especially helpful owing to its size, complexity, required build environment, etc.
- A program that uses multiple executable components is inherently concurrent, but modern components do not provide many functions for concurrency control.

3 EXPERIMENTAL SYSTEMS APPROACH

The key aspect of our experimental method is to take as a stated objective to create an industrially competitive system that is representative of a broad class of related systems.

We adopted the “industrially competitive” goal not because we want to sell a product, or because our research is short-term, but because unless such an objective is accepted it is too easy to miss problems that will be encountered in practice. Adopting a “product” goal, rather than building a toy system, is thus important to avoiding “false positives” in an assessment of the potential of the POP model. False positives have generally been the problem in many earlier assessments of CBSD models.

We took the goal of building a system that is representative of a broader class of systems in order to have a basis for deriving generalized results. As we discuss in more detail in the next section, we selected computational modeling and analysis tools for engineering as a general domain; and we selected as a particular application within that domain, dynamic fault tree analysis, for our case study. We selected dynamic fault tree analysis primarily because we believed that there would be industrial demand for an application in this area; and our evaluation approach requires such demand to operate. We also had access to local expertise in computational reliability engineering.

Although one system can only cover a limited subset of the issues that will be faced in practice, we do believe that building an industrially competitive tool forces us to face some of the demanding requirements that arise in practice. In order to improve our chances of understanding what is needed, we distribute prototypes of our software to all interested parties on the World-Wide Web. We have interacted with industrial users to gain feedback on shortcomings, leading to requirements for subsequent iterations. Based on work to date, we are building a version of the tool intended to be used in practice, under an agreement with NASA Langley Research Center.

4 MODELING AND ANALYSIS TOOL DOMAIN

Engineering modeling tools are sold in niche markets. It is hard for software companies to amortize development costs across a large number of users. Yet tool users demand sophisticated model manipulation capabilities, as well as rich functions such as graphical zooming, printing, cut-and-paste, report generation, spell checking, etc. Providing this kind of function from-scratch development is very costly, resulting in tools of on the order of a million lines of code [22]. The upshot is that sophisticated tools, if built at all, carry high prices or less functionality.

1 Galileo

We (and others) have observed that many tools comprise an analysis core plus a surrounding superstructure supporting many such costly functions; and that modern mass-market packages provide much of the needed function. Thus the question naturally arises, can we apply the POP CBSD model to avoiding having to build costly superstructure for sophisticated software tools for engineering?

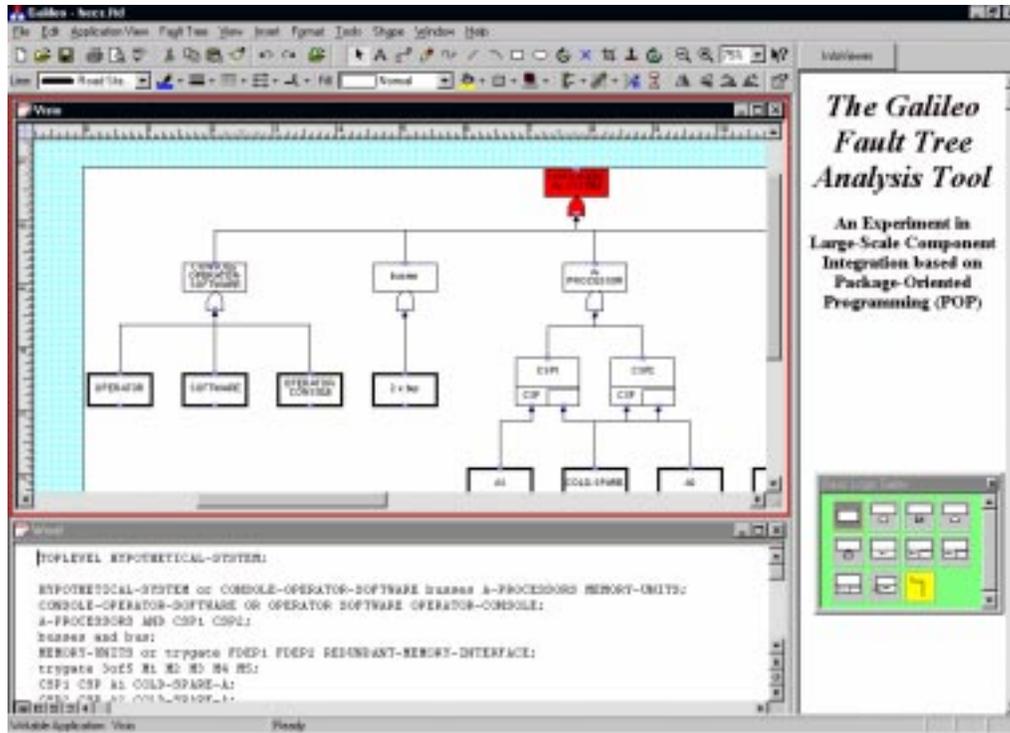


Figure 1: Screenshot of Galileo 2.11a

For several years we have been evaluating POP by building a dynamic fault tree tool, Galileo [19,20,21,22]. A fault tree [26] is a system model used to estimate system failure probabilities. A fault tree is a directed acyclic graph whose bottom nodes are basic events and whose internal nodes are gates. Each basic event and gate corresponds to a failure. A basic event models the failure of a system component by giving its probability of occurrence. A gate specifies a subsystem failure as a function of its input (basic event or other gate) failures. The top-level gate corresponds to a system failure. Fault tree analysis computes the probability of the top-level event as a function of basic events probabilities and the structure of the tree. Traditional fault trees models represent only time-independent relationships between input and output failures (such as AND and OR), but more recent work (DIFtree) has incorporated order-dependent gates [2,6] and modular solution methods [7,13].

Galileo support modular dynamic analysis through the user interface presented in Figure 1. The upper-left subwindow displays the graphical view of a fault tree, provided by Visio Corporation's drawing application Visio Technical. The subwindow below displays the equivalent textual view, provided by Microsoft Word. The right side displays a help system implemented by Microsoft's Internet Explorer. It also shows the graphical "stencil" that holds shapes for use in the graphical construction of fault trees. All of these subwindows are integrated within the Galileo window.

Some of the main features of Galileo are:

- full support for the DIFtree analysis approach
- multi-view environment for building fault trees
- textual fault tree programming language
- graphical fault tree language with multi-page support
- ability to edit textual and graphical representations
- automatic update of textual and graphical views
- user interface built from widely known packages
- integration of package windows into a single interface
- on-line documentation through integrated browser

2 The Architecture of Galileo

Figure 2 shows the basic architecture of Galileo. The main Galileo mediator coordinates the views and the analysis engines, and implements the main application window. Most of the user interface and modeling function of the tool is provided by the package components. Each component—Visio, Word, and Internet Explorer—runs as a separate process (indicated by dashed boundaries), but is integrated in the same user interface.

Galileo has two methods for editing a fault tree: a graphical editor and a textual editor. The user creates a fault tree in the graphical view by moving fault tree shapes from the stencil to the drawing page, and then connecting them to

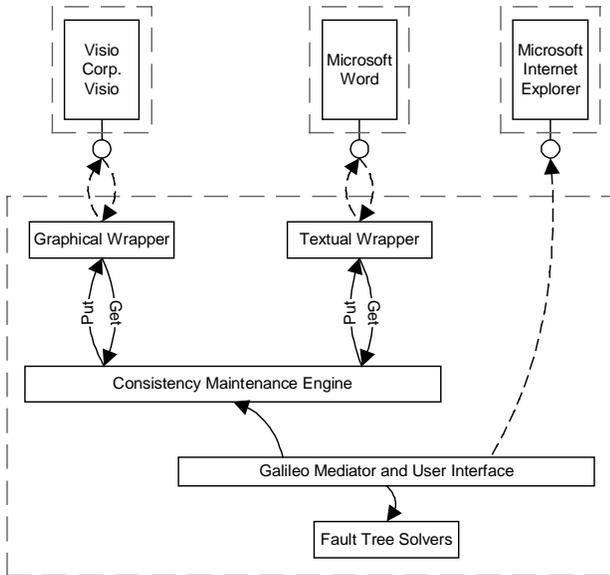


Figure 2: The high-level architecture of Galileo

express the relationships between the shapes. The textual representation contains the same information as the graphical. It can be edited like any text document. In both views the user can use the standard editing functions of the component applications. The graphical view is more intuitive, but some users find the textual view faster to use.

The textual and graphical interfaces are implemented by Microsoft Word and Visio respectively. The packages are encapsulated by wrappers, each of which implements two key functions, *get* and *put*. The *get* operations extract a fault tree object (C++ class instance) from a view, provided that the view depicts a valid fault tree. The *put* operation works in reverse, rendering a fault tree as either a text stream or a graphical drawing. Consistency is implemented using a simple scheme based on cache coherence. The algorithm does a *get* on the active view followed by a *put* on the view to be made active. A fault tree acquired from either view can be sent to the analysis engine.

Galileo presents views only for those package components that are actually installed on the user's system. A user who lacks Visio sees only the text view based on Word, for example. A user can edit the currently active view, browse both views, and render the active view to the other view for editing. The textual and graphical views are saved as a single file. Galileo manages component initialization, lifetime, shutdown the invocation of underlying analyzer.

Our ability to specialize and integrate the components was critical. For example, we customized menus to keep users from closing components independently. We created a Visio *stencil* containing fault tree shapes, and programmed

Word to disable grammar checking for fault tree documents. We also customized Internet Explorer so that it displays Galileo hypertext documents, but does not act as a general web browser.

We used Microsoft's OLE to drive the components through their application programming interfaces. We used the Microsoft Active Document standard [16] to integrate the interfaces of the packages into the Galileo interface. The components conform to Microsoft's underlying Component Object Model (COM) standard [18], which provides for low-level remote procedure call interoperability. In Figure 2, dashed arcs represent function calls across application boundaries, and solid arcs, normal function calls.

The Active Document standard was particularly important. It made it possible to integrate multiple application interfaces into a coherent presentation. It supports merging of menus of separate applications, and their presentation as sub-windows of Galileo. As views are selected, buttons, menus, and other user interface components change to reflect the package interface for that view. Galileo provides its own menu items, which are always present. One allows the user to invoke the consistency protocol. The other invokes fault tree specific actions such as the analyzer.

5 EXPERIENCE IN DEVELOPING GALILEO

We have developed Galileo based on the needs of industrial users. We developed an early version largely based on a technical evaluation performed by an engineer at Lockheed Martin, who helped clarify what was present and missing from the tool in terms of usability, modeling capabilities, and analysis capabilities. Engineers and researchers at NASA Langley Research Center (LaRC) also provided feedback on the value of aspects of our early system. The evaluation given by the Lockheed Martin engineer in an internal report was that Galileo has significant potential to aid reliability engineering at the firm. NASA LaRC funded the development of a production version.

From the beginning we have released intermediate versions of the tool for free download off the WWW. To date many hundreds of industrial users have acquired the tool. This strategy has also been a source of valuable user feedback. One piece of feedback from a major defense contractor described the inadequacy of the user interface in its lack of support for multiple page and hierarchical drawings. We took that feedback as the basis for the requirements for the enhanced user interface to be provided in the final NASA version (which is to be delivered in 2001).

1 Component capabilities

The original suite of components that we identified as providing leverage for Galileo were Microsoft's Word and Access, and Visio's drawing tool. (At the time, Visio was owned by Visio Corporation. It has since been acquired by Microsoft.) We observed that Word and Visio covered a

significant portion of the required functions, and that they had mechanisms for specializing them for our application. Access provided basic persistence and concurrent access to multiple fault trees, and the ability to generate reports.

We encountered difficulties with Access as we migrated from our early version, in which package windows were distributed around the user's screen, to an architecture based on the Active Document standard. Access did not support that standard, even though it is a member of the Office suite. The lesson is that conformance of components to standards cannot be taken for granted. We stopped using Access, and instead turned to the file system for fault tree storage. A second lesson is that a willingness to search for solutions that provide value is of the essence in this development approach, in which one continually learns about unexpected aspects of the development environment.

Not all surprises are negative. When Microsoft's Internet Explorer became available as an architecturally compatible component, we saw the opportunity to integrate it into Galileo to provide a hypertext-based on-line help system.

2 Examples of iterative design exploration

We have found the process of developing a system in this style to be one of continually trying to reconcile an understanding of what the user will value with the capabilities and constraints imposed by the component base. Traditional top-down system design is ill suited for developing applications using this approach [5]. Rather, the early stages of design resulted in an architecture within which we continually search for workable designs. We now discuss several dimensions of this basic point.

1 Batch-oriented editing

Early in the design we discovered that Visio did not expose the "delete" event adequately, which hindered our ability to implement an eager, incremental update scheme. Consequently, we could not detect fine-grained editing operations, which caused us to move to a batch-oriented consistency scheme that follows a compiler-like pattern [22]. The user creates a fault tree representation and then "compiles" it to be analyzed or rendered in another view.

The resulting batch-update protocol that we designed for maintaining consistency between views requires the user to put the active view in a consistent state before the model can be rendered in the alternate view. Any changes made to the inactive view are lost when the consistency maintenance system is invoked.

We had hoped to be able to make the inactive view read-only, but found that that the applications did not support such a function. We were able to prevent window events from reaching applications to make them "read-only" but this approach also disabled document navigation functions. Because of this limitation, we modified our design such

that the inactive view is highlighted in red, but editing it is not disabled. The solution is somewhat unintuitive and it increases the mental effort on the part of the user. The question is whether the benefits are enough to outweigh such inconveniences. We do not fully know the answer at this time.

2 Multiple-page support

Our earlier versions of Galileo were limited to drawing trees on one page. A potential user emphasized the need for multi-page drawings. This new requirement placed demands on the Visio component. To mitigate the risk that we could not manipulate Visio pages programmatically, we built a standalone, non-Active Document prototype that created a Visio process and created multiple page drawings.

During later development we discovered to our surprise that the page manipulation functionality provided by the published API did not work: precisely when Visio was used as an Active Document! Its page manipulation interface did not function when a document appeared as a child window in another program. Unfortunately, we could find no workaround. The firm requirement of our "customers" to have this support caused us to contact the component vendor to lobby for correct support of the published API in the next version of the package.

During our discussions with Visio Corporation, it became clear that to get the functionality we needed we had to provide a rationale in business terms. After we described our funding from NASA Headquarters under our agreement with NASA LaRC, they agreed to fix the problem in the next release. We continued to develop our tool to use the page manipulation functions, in the expectation that a suitable version would become available, which it did. We found this problem only because we were both constructing a system from multiple components, and because we were responding to real requirements from real users.

3 Automatic layout

We also used prototyping to explore the feasibility of using Visio's built-in layout engine to automatically layout the fault tree during model construction. This feature was one that a potential customer told us distinguished high-quality tools from the others. Our implementation at the time computed the layout of the graphic shapes, and caused Visio to position each shape one at a time. This resulted in extremely slow layout of the fault tree. Success with the prototype gave us confidence that we could utilize Visio's efficient built-in automatic layout function, avoiding cross-application communication costs. The resulting increase in layout speed meant that it would be possible to invoke the layout algorithms after every high-level editing operation.

4 High-level fault tree editing operations

Earlier versions of Galileo provided only built-in Visio functions for graphical editing of fault trees. For example,

to connect two gates, the user would have to drag two gate shapes from the stencil to the canvas, then drag a connector shape to the canvas, then link each end of the connector to each of the shapes. Our potential users complained that this editing interface was almost uselessly tedious. We decided to provide a set of fault tree drawing construction functions to automate most of these tasks. For example, we provided a function that allows a user to select one gate and then with the push of a button to add and connect a gate under it.

We explored three different sets of high-level operations, attempting to identify useful operation sets that could be implemented efficiently given the constraints of Visio. For example, no operation could be specified that required iteration over every shape in the drawing, due to the cross-application speed constraint.

Having settled on what appeared to be both valuable to users and feasible given the component constraints, we needed to actually implement the functionality on top of the only partially known Visio virtual machine. We mitigated the implementability risk using a “bridging” approach. We decomposed the high-level, domain-specific operations into more basic low-level operations. Simultaneously, we performed a bottom-up abstraction of the raw Visio API calls into more domain-specific operations. The condition that these two efforts “meet in the middle” had to be satisfied before we could proceed to implementation. It took three attempts before we found an operation set that we could implement effectively.

5 *Cross-page linking*

A new graphical interface feature we added was the ability to link a connector on one page to a sub-tree on another page. We tried representing bi-directional links by storing Visio unique IDs of linked shape within the linking shapes. A sub-tree linked from many places would thus store such an ID for each such location. Unfortunately, all of the available methods for storing such references placed severe limits on the number of references that could be stored, which would have prevented us from providing the scalability that users desired. Furthermore, we discovered through experimentation that the references would be invalid if the referred shape was “cut” and then “pasted” from the drawing page, because the unique IDs changed.

When we contacted Visio Corporation, they told us that the upcoming version would have hyperlinks that would give us the functioning we needed. We continued development assuming that the functions would be provided. They were. In this case, component evolution coupled with knowledge from the component vendor helped us to determine a workable design. The lesson is that design in this style is anticipatory in an interesting way: we were taking risks on future features and fixes that were not assured.

6 *Augmenting the graphical interface*

One of Visio’s specialization capabilities is to allow the developer to create custom buttons on the user interface that invoke custom code. Unfortunately, we discovered that although we could use this capability to invoke code using Visio’s Visual Basic for Applications, we could not cause our Visual C++ code to be invoked; and that is what we needed to provide high-level fault tree editing operations to users. As a workaround, we chose to modify our design so that the buttons were owned by the main Galileo window, and that the activation of the buttons would cause Galileo to invoke the necessary functions on Visio. The downside of this design is that the buttons do not disappear when Visio is not the active view, as they would if they were being managed with the rest of Visio, as an active document.

In this case, our inability to make the original design work might be at least attributed to our lack of knowledge concerning the proper method of invoking external code. Given our uncertainty in whether the problem was our lack of knowledge or an inability of the component, we opted to use an alternate design which was feasible and low cost, if not entirely satisfactory.

7 *Unexpected behaviors in the virtual machine*

In addition to prototyping, our team, led by Copenhafer, used automated techniques to verify performance and capacity assumptions of package components. He used an automated tool to systematically explore performance, and to verify the performance of new component versions [5]. During these experiments we found yet another unexpected and seemingly arbitrary component characteristic. In order to speed up drawing, we disabling view rendering while adding shapes to the page. We found to our surprise that view rendering was re-enabled whenever we created a new drawing on which to place shapes. Thus, in order to insure view rendering is disabled, one should be sure to disable it *after* creating any new drawing.

8 *Capacity limitations of Visio*

Examples of capacity limitations include a limit of 31 characters on the length of shape identifiers, a limit of 254 characters on a “list” custom property of a shape, a limit of 127 characters for a “string” custom property, and a limit on the number of handles to Visio shapes that the programmer can hold open at one time. Not all of these limits were documented.

9 *Evolution of components*

Our experiences dealing with component evolution have been generally positive, despite the negative concerns that many researchers have expressed about the evolution of components. Our architecture accommodates evolution by encapsulating the details of interaction behind the view wrappers. It also supports multiple versions of the same component.

The components themselves have generally improved over time in terms of the repair of known problems, and in the degree of user-level functionality that was exposed programmatically to the developer. We have already described how Visio improved. We had similar experience working with Microsoft Word. The first version we used, Word 6.0, had a narrow interface consisting of a single method through which Visual Basic commands were invoked. Our efforts to integrate this component revealed a race condition. Our design used `wdEditSelectAll` to select the whole document text, and then it retrieved the selected text. However, if we performed the second operation immediately after the first, we would not get all of the text. We found that we could avoid that malfunction by extracting the text after waiting 500 milliseconds for the selection operation to complete.

The next version, Word 95, with a full OLE interface, did not have such a race condition. Unfortunately, we found that calling the `Selection()` operation followed by `GetText()` failed because the selection size had a limit much lower than the documented upper limit of 65,000 characters. The alternative design we identified and implemented was to copy the text to the Windows clipboard, and then read the clipboard to acquire the entire text of the document. The next version, Word 97, had none of these limitations, and we were able to implement our functionality without these awkward workarounds.

For the Word upgrades from 6.0 to 95, and from 95 to 97, and the Visio upgrades from 4.0 to 4.1, and 4.1 to 5.0, we simply installed new versions of the packages on our machine, ran Galileo, and found that we had an updated tool. The only upgrade incompatibility we observed was the upgrade to the new Visio 2000, which failed presumably because the new API is not backward compatible. We have not yet verified the cause of this failure.

There are several lessons we learned in this experience. The first is that components such as Microsoft Word seem to be improving over time. Second is that while components are improving, they will always have extremely large interfaces that are likely to be undocumented, or misdocumented. The components are also complex software entities that may have unknown (even to the vendor) timing or capacity limitations that can cause a particular design to be difficult or infeasible.

10 Evolution of user interface architecture

The need to integrate the programmatic interfaces of components is obvious. What is perhaps not as obvious is the need to integrate their user interfaces. An early version of our tool did not integrate the interfaces of the component applications, but presented multiple package windows to the user. In addition to causing usability problems, such

loose integration allowed the user to violate system-wide invariants. For example, the user interfaces of the component applications provided several mechanisms for closing the application component, which would cause Galileo to fail when it later tried to access the component.

Our initial efforts to design around these limitations involved working outside the programmatic integration architecture provided by COM. We developed a technique for modeling the internal behavior of a component (because it is not always exposed), monitoring the user's actions at the operating system level, and then preventing those actions that violated system invariants [15].

This approach is difficult and non-intuitive, but is the only option when the component does not provide sufficient "hooks" for intercepting actions. Goldman and Balzer [12] had a similar experience building a domain-specific design environment on top of Microsoft Powerpoint, where they had to poll the Windows event queue to infer editing operations that were not exposed by Powerpoint.

6 EVALUATION OF POP

We believe that we have framed a set of requirements for Galileo satisfaction of which will result in an industrially viable tool, and that we have mitigated most of the major risks presented by our component packages. We are still distributing an early version of the tool that limits users to one-page drawings, yet we have distributed the tool to many hundreds of industrial users and interest in the tool continues to grow. Anecdotally, users find the underlying modeling and analysis function combined with the ability to use standard PC tools for editing and presentation to be quite valuable.

We believe that our tool is representative of a broader class of modeling and analysis tools for engineering. Informal surveys of tools for system performance modeling, software architecture, and other such applications shows a common pattern of support for graphical and textual notations and standard user interface functions combined with an underlying set of analysis algorithms. To the first order, Galileo appears to be a reasonable representative of this class of applications.

Our experience appears to indicate that with enough effort it is possible to leverage the capabilities of mass-market packages to build such applications at a cost that is small-constant-proportional to the cost of their relatively small core modeling and analysis cores. In other words, there is evidence that POP defines a viable model for CBSD, at least for such tools—one that can produce many orders of magnitude in cost reduction compared to a build-from-scratch approach.

Important factors working in favor of POP include volume pricing of dual-use (application and component) packages;

widespread use of components, with associated, existing licensing and payment model; de facto decomposition of the application space into general, orthogonal subdomains, such as technical drawing and text editing, that are likely to be needed across wide variety of systems; and the provision of both APIs and user interface integration architectures.

Several of these features place the POP model near that of the successful model of operating systems, data bases and web browsers that Lampson identified. The high development costs associated with packages are offset by volume pricing. The widespread use of such applications means it is not unreasonable to assume that they are available as machines on which to build applications. And they address important, general problems that are relevant across a variety of systems.

On the other hand, using POP with today's components is fraught with difficulties. The components are complex and rife with undocumented behaviors and limitations. We had to use a development style based on prototyping and the ongoing exploration of both component properties and user requirements. The risks are very high. We were caught several times late in the game with serious problems that threatened to undermine our project. In particular, if Visio had not come through with both adequate support for hyperlinks and fixes for the multi-page aspects of the API, we would not have been able to implement the multi-page, linked drawing functions that our users required.

The bottom line based on our evaluation is that that POP is a CBSD model that is well worth continuing to pursue. The model instance available using today's components can provide very considerable value. However, anyone who considers adopting it must be aware that the risks are considerable. There are likely to be potholes, possibly some very deep ones, along the way.

7 RELATED WORK

The feasibility of the package-oriented approach was first discussed by Sullivan and Knight based on a very early prototype that did not meet real user requirements [22]. Since then, we have gained experience in the extensive use of application components.

Goldman and Balzer report on the use of Powerpoint as a platform for building a software architecture modeling and analysis tool [12]. They cite many of the same benefits that we discussed here and in earlier work. Importantly, we have explored the integration of multiple components using the POP approach. Our tool was also driven by demanding customer requirements. We have also addressed the component evolution issue.

In the context of U.S. government COTS initiatives, Brownsword et. al describe the impact of components on lifecycle activities such as requirements definition, testing,

and maintenance [4]. They also identify the relationship between component selection, requirement specification, and architecture design, and the potentially high cost of specializing components. They provide no real examples.

Fox et. al provide a COTS-based software development process for information system infrastructure [9,10]. As in our work, there is a heavy emphasis on early component evaluation and selection, and risk reduction methods such as prototyping. They also cite the problems of understanding large and complex components, and the strategy working with components given incomplete knowledge. However, our experience is that problems of component upgrade might not be as bad as they describe.

8 CONCLUSION

We continue to search for successful CBSD models, and for a better understanding of the conditions under which such models are likely to succeed. We have presented an evaluation of one promising model, based on the use of architecturally compatible mass-market software packages as components. Our evaluation approach is based on the best effort use of a model against demanding, industrial requirements: on building real, not toy, systems. We found that the model has enormous potential, and that even with the today's components, significant value can be created; but that today the model still involves very significant risks.

ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grants MIP 95-28258, CCR-9502029 (CAREER), CCR-9506779, CCR-9804078 and by NASA Langley Research Center and Ames Research Center. We thank reliability engineers at NASA Langley Research Center and Lockheed-Martin Corporation for their valuable feedback.

REFERENCES

1. Ted Biggerstaff and Charles Richter. Reusability framework, assessment, and directions. *IEEE Software*, 4(2):41-9, March 1987.
2. 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.
3. Fredrick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*, 20th Anniversary Edition. Addison Wesley, Reading, Mass., second edition, 1995.
4. Lisa Brownsword, David Carney, and Tricia Oberndorf. The opportunities and complexities of applying commercial-off-the-shelf components. *Crosstalk*, 11(4):4-6, April 1998.

5. Michael A. Copenhafer and Kevin J. Sullivan. Exploration harnesses: Tool-supported interactive discovery of commercial component properties. In Proceedings of ASE-97: The 12th IEEE Conference on Automated Software Engineering, Cocoa Beach, Florida, 12-15 October 1999. IEEE.
6. 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.
7. 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.
8. E. Feigenbaum, Chief Scientist, U.S. Air Force, IEEE Computer, January 1998.
9. Greg Fox and Steven Marcom. A software development process for COTS-based information system infrastructure: Part 1. Crosstalk, 11(3):20-25, March 1998.
10. Greg Fox, Steven Marcom, and Karen W. Lantner. A software development process for COTS-based information system infrastructure: Part II lessons learned. Crosstalk, 11(4):11-13, April 1998.
11. David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. IEEE Software, 12(6):17-26, November 1995.
12. Neil M. Goldman and Robert M. Balzer, The ISI Visual Design Editor Generator. In 1999 IEEE Symposium on Visual Languages (VL'99), Tokyo, Japan, September 1999. pp 20-27.
13. 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.
14. Butler Lampson. How software components grew up and conquered the world. In Proceedings of the 21st International Conference on Software Engineering, Los Angeles, California, 16-22 May 1999. IEEE.
15. Mark Marchukov and Kevin J. Sullivan. Reconciling behavioral mismatch through component restriction. Technical Report CS-99-22, Department of Computer Science, University of Virginia, 30 July 1999.
16. Microsoft. "Active Document Containers." URL: http://msdn.microsoft.com/library/devprods/vs6/visualc/vccore/_core_activex_document_containers.htm
17. James M. Neighbors. Draco: A method for engineering reusable software systems. In Ted J. Biggerstaff and Alan J. Perlis, editors, Software Reusability --- Concepts and Models, volume I, chapter 12, pages 295-319. IEEE, 1989.
18. Dale Rogerson. Inside COM. Microsoft Press, 1996.
19. 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.
20. 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.
21. 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.
22. 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.
23. Kevin J. Sullivan, Mark Marchukov, and John Socha. Analysis of a conflict between aggregation and interface negotiation in Microsoft's Component Object Model. *IEEE Transactions on Software Engineering*, 25(5), September 1999.
24. Clemens Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 1998.
25. Jon Udell. Componentware. Byte, 19(5):46-56, May 1994.
26. United States Nuclear Regulatory Commission, Fault Tree Handbook, NUREG-0492, 1981.