Exploration Harnesses: Tool-Supported Interactive Discovery of Commercial Component Properties

Michael A. Copenhafer and Kevin J. Sullivan
University of Virginia Department of Computer Science
Thornton Hall, Charlottesville, VA 22903 USA
{mac3b, sullivan}@cs.virginia.edu

Abstract
A key problem in component-based software development (CBSD) is that developers have incomplete knowledge of components. In many cases, the only available source of such information is experimentation. In this paper we argue that the provision of tool support for automated and repeatable experiments can provide significant value to designers. Such tools, which we call exploration harnesses, promise to help enterprises manage and exploit component evolution. We evaluated the exploration harness concept by building a prototype and using it to support the exploration of components in the design of a fault-tree analysis system called Galileo. Galileo employs package-oriented programming, in which shrink-wrapped packages such as Microsoft Word and Visio Technical are used as large components. Using our exploration harness helped us to discover a range of undocumented properties of such components, across several versions, which enabled us to make better informed design decisions.

1. Introduction
Component-based software development (CBSD) is the design of software with parts obtained in a commercial marketplace [1, 2, 3, 4, 11]. Emphasis shifts from programming to component integration. CBSD promises improved cost, time, and quality. Yet consistent success remains an elusive goal. One problem is that the designer often has incomplete knowledge of key properties of given components. The late discovery that it is hard to integrate components in the desired manner [5] is but one possible consequence of incomplete knowledge. More generally, incomplete knowledge makes it hard to make the most effective design decisions with confidence: whether to use given components in the first place, and, if so, how to use them to best effect.

Incomplete knowledge of component properties is of course nothing new to software designers. In particular, it is not a problem new to component-based design. If complete knowledge of properties of software parts was easy to obtain, e.g., given source code, there would be no need for such activities as testing or inspection. Manifestly, incomplete knowledge is an essential condition under which designers have always struggled, and for which there can be no entirely satisfactory solution.

However, the shift to designing software by integrating larger commercial components complicates the problem in significant ways, creating the need for novel approaches to dealing with the problem. First, the designer qua integrator does not control component development and so lacks first-hand knowledge, e.g., of performance properties and limitations, or of architectural assumptions. Second, the integrator typically has limited access to those who do have such knowledge. Component designers operate behind corporate boundaries and are compelled to avoid the unplanned disclosure of potentially valuable information. Third, publicly available information is almost always incomplete, if only for intellectual property protection. Fourth, changes in properties of components that affect the design space can occur from one release of a given component type to another. Fifth, the complexity of components can hide relevant properties. The problem is worse for large components such as shrink-wrapped application packages.

It is important to note that although incompleteness of knowledge is generally seen as creating problems for the designer, it also establishes a critical space for value creation in a competitive design environment. In particular, a designer can gain advantage against competitors by augmenting his or her own private knowledge of key component properties, e.g., through experimentation. Competition comes to be based in part on the ability of designers to deepen their knowledge of the materials at hand. Designers with better abilities are thus able to make better design decisions.

In the absence of first-hand knowledge or complete and accessible documentation, the designer has little choice but to learn by experimenting upon components. The contribution of this paper is the idea that automated software tool support for repeated, systematic interactive experimentation upon components under a range of
operating conditions has the potential to provide significant value to component-based software designers. We call such tools exploration harnesses. To evaluate the idea we designed and implemented a prototype exploration harness and used it to explore properties of shrink-wrapped application packages that we are using as components in a fault-tree analysis system called Galileo. We discuss the design of our tool, and present results obtained by using it. Our experience suggests that exploration harnesses have the potential to be especially valuable in developing knowledge of new versions of components.

Section 2 discusses Galileo, focusing on problems that we encountered in integrating its component packages. Section 3 discusses the exploration harness concept and how we used it in designing Galileo. Section 4 evaluates our work to date. Section 5 discusses related work. Section 6 concludes.

2. Galileo

This section presents a brief overview of the component-based system in which we both developed and employed the exploration harness concept. We focus on difficulties that we had in integrating Galileo’s package-based components in order to highlight consequences of incomplete knowledge and the need for ways to learn effectively. In the next section, we introduce exploration harnesses and explain how we used our prototype tool to ameliorate some of these problems. Galileo employs a design style that we call package-oriented programming (POP). We use multiple shrink-wrapped application packages, such as Microsoft Word and Visio Technical, as components. More complete discussions of package-oriented programming and Galileo are available elsewhere [6, 8, 10].

2.1 Project overview

Galileo consists of a small computational core for fault-tree analysis hosted by a superstructure that supports text-based and graphical rendering and editing, web-based documentation browsing, persistent storage of fault-trees, and other such functions. In order to develop a rich superstructure having a sophisticated and familiar look-and-feel quickly and at low cost we decided to use volume-priced shrink-wrapped application packages as components. The superstructure uses three packages: Microsoft’s Word and Internet Explorer and Visio Corporation’s Visio Technical. The packages are integrated into a single window by an “active document” container. The container coordinates the windows and menus of the component packages, and provides functions for maintaining the consistency of read-write graphical and textual views of a fault-tree in the Word and Visio components, respectively.

2.2 Development problems

The problem of incomplete knowledge is worsened for components, in general, and when developing with complex components, such as major shrink-wrapped packages, in particular. These components are large, very complex, not documented rigorously, and are subject to use in unexpected ways. As a result, in our experience, they have undocumented properties that are relevant when they are used as components. Furthermore, these packages evolve through multiple versions. Newer versions can have properties that are both better and worse than those of their ancestors. In this paper, we summarize problems that we had using these packages as components in five areas: (1) evolution; (2) inadequate capabilities and performance; (3) inadequate documentation; (4) limited access control and concurrency control; and (5) hidden shortcomings.

2.3.1. Component evolution. During the four-year lifetime of Galileo, the component packages that we are using have evolved significantly. Newer versions of the Microsoft Office applications support a unified programming interface including an object hierarchy and embedded Visual Basic programming environment, Visio Technical added support for use as an active document in an active document container. Older versions of Galileo have invariably worked with new component versions. As a result, new components, in our experience, have offered an instant, low-cost upgrade path. However, we found it valuable to change parts of Galileo to exploit the new properties of the latest package versions.

In general, we have struggled with shortcomings in component packages. We have had to make compromises in the design of Galileo to work around these shortcomings. We track the evolution of the packages in these dimensions with care. We have continually upgraded Galileo to take advantage of new capabilities as they appear. One problem that we are now grappling with is the inability of Visio 5.0 to handle multiple-page drawings in an active document container. We are employing an awkward work-around at present, but we anticipate having the desired support in a forthcoming version of the package.

2.3.2. Performance and capabilities. In earlier work we described some shortcomings of Galileo’s packages [6,
In this paper, we highlight performance issues. Galileo’s response time, for example, is largely dependent on package performance. An especially difficult problem for us has been the drawing performance of the current (5.0) and earlier versions of Visio. For example, Visio Technical version 4.1 on a 133 MHz Intel Pentium with 64 MB RAM took approximately 22.5 minutes to render a fault-tree with 2,000 gates: 6.5 minutes to draw the gates, and 16.0 minutes to connect them. Galileo remains useful because the user has the option to edit fault-trees in textual form, and to render them once when needed; but at this speed, high-frequency switching between the text and graphical views is not practical. We spent significant effort understanding Visio’s drawing speed and exploring ways to use Visio that would give us the best drawing performance.

2.3.3. Inadequate documentation. Most of the packages provided limited documentation of programming interfaces. Microsoft Word and Access, for example, provided little documentation on how to use their type libraries in conjunction with C++. The precise behaviors of documented functions were also typically not documented. Additional information on program bugs, limitations, and performance was also beyond our reach. In some cases, published documentation on package limitations was incorrect.

2.3.4. Limited access and concurrency control. We found that the packages did not provide adequate facilities for restricting the range of functions accessible from the user interface or for mutual exclusion of user and program manipulation. Thus, in some versions of Galileo users could execute package functions (such as exit) that were inconsistent with the needs of the Galileo tool.

2.3.5. Hidden shortcomings. Due to lack of package documentation, we discovered certain limitations of the packages only after working with them extensively. In some cases, the limitations were quite serious. For example, we found that Visio 4.1 and 5.0 malfunctioned if a client program (such as Galileo) held more than about 700 active references to Visio drawing objects. This was a problem since our design assumed that the number of references was limited by the amount of memory. A workaround was possible: each shape has a unique identifier, so we held these identifiers and looked up shapes on demand. The cost was additional runtime performance overhead.

Our recent efforts have been directed at improving the Galileo user interface. Earlier versions had multiple package windows on the screen. Users complained about the management of the views being tedious and error-prone. Our current version uses Microsoft’s ActiveX Document technology to manage the packages as child windows. This makes switching between views easier and increases our ability to restrict certain features of the packages. Users can no longer accidentally close the Word and Visio windows. However, to our dismay, we later found that certain key properties of Visio were not available while executing as an ActiveX Document. For example, we could no longer program Visio 5.0 to create multi-page documents. Drawing large fault-trees on a single page is not viable for our users.

3. Exploration harnesses

Faced with incomplete knowledge, the designer has little choice but to learn by experimentation. One way to do this is to try to build the desired system and to learn along the way. To some extent the need to do this is unavoidable. However, it can be an extremely costly approach, both in terms of risks that do not materialize until it is too late, and in terms of missed opportunities. We believe that in some cases, there is a better way. The designer can obtain considerable advantage by exploring packages interactively, systematically, and under a variety of conditions with automated tool support. Tool support is especially valuable when the same experiment needs to be done under varying conditions, e.g., with the package in different states or with different versions of a package. Repeatability will be critical for dealing with component evolution. We now describe the design and use of a prototype exploration harness.

3.1 Description of our prototype

Figure 1 illustrates our prototype exploration harness’s component architecture. The system consists of three major elements:

1. Harness: The harness provides a graphical user interface for selecting and executing “exploration cases.” An exploration case, in analogy to a test case, defines how a component is to be manipulated dynamically and what information is gathered. The harness also has the ability to gather system information such as operating system version, processor speed, and memory usage.

2. Exploration Component: An exploration component contains a set of exploration cases for a particular package. We created a test component for each of Galileo’s packages, for example. A test component is a COM component that implements a single COM interface, ITest. This interface allows a client program—the harness in particular—to access methods of the test component to execute test cases.
3. Report Generator Component: The report generator component is a COM component that implements two interfaces: \textit{IReportGen} and \textit{IGraphGen}. These two interfaces support data display in numerical and graphical format, respectively. The report generator uses Microsoft’s Automation technology to programmatically control Microsoft Excel 97 (used as a component) to generate graphs of collected data.

![Figure 1. Architecture of exploration harness](image)

**Figure 1. Architecture of exploration harness**

We designed and implemented the exploration harness system in a few person weeks using Microsoft Visual C++ 6.0 and the Microsoft Foundation Classes. The harness is about 1200 lines of code and the report generator component is approximately 500 lines of code.

Figure 2 shows a screen shot of the user interface. The main user interface element is a list control containing the current set of exploration test cases. Progress bars display memory information and test progress.

![Figure 2. Screen shot of exploration harness](image)

**Figure 2. Screen shot of exploration harness**

3.2 Using the exploration harness

To explore a package, the user first packages exploration cases in an exploration component and registers it in the Windows Registry. After registering at least one such component, the exploration harness is used as follows:

1. Run the tool – a list of registered exploration components is displayed.
2. Select a test component.
3. Select an exploration case.
4. Change exploration case parameters such as the number of iterations.
5. Select and parameterize additional exploration cases, if desired.
6. Establish operating conditions of component under test.
7. Execute all or a selected portion of the exploration cases set.

The harness automates exploration case execution and data collection. The operator can define and execute multiple exploration cases under varying conditions. Conditions that might vary include whether the screen update mode of a package component is enabled; whether memory is in use by other programs; whether files reside on local or network drives; etc.

4. Evaluation

We now discuss the general format of the exploration cases we used to explore properties of Galileo’s packages. We focus in particular on a set of exploration cases that revealed unusual properties or limitations of the packages, and that proved particularly helpful in enabling us to make decisions as components evolved. For each case, we describe how the results influenced the design of Galileo.

4.1 Exploration cases

Because the drawing performance of Galileo was critical and inadequate, much of our exploration focused on this issue—and thus on properties of Visio, in particular. Many of our exploration cases were designed to help answer questions such as this: "How long does it take to manipulate $n$ drawing elements as a function of the size of the drawing." For example, how long does it take to draw one hundred Visio shapes as a function of the number of shapes already present? This question was important in reasoning about the performance of incremental updates to Visio drawings. We also created test cases to explore package limitations. For example, what is the maximum number of shapes that can appear in a Visio drawing?

Our first step was basic performance benchmarking. We then explored the conditions under which we could obtain improved results by repeating the exploration cases while varying a number of operational parameters. The following is a partial list of these parameters:
• **Client type:** (1) Executable OLE client, (2) VSL (a user extension of Visio in the form of a dynamic link library), (3) VBA macro (Visual Basic for Applications, Microsoft’s macro language for Office 97 products).

• **Window/View Configuration:** (1) Viewing area within clipping region or not, (2) Window iconified or maximized, (3) Screen updating (refresh) on/off.

• **Operating System:** Windows 95, 98, NT.

• **Operating Platform:** (1) Processor speed, (2) Amount of physical memory, and (3) Available disk space.

• **Package Version:** Visio 4.0 and 5.0, for example.

### 4.2 Sample results and conclusions

Space limitations prevent us from presenting detailed results. We discuss four cases for which the data benefited or influenced the design of the Galileo system. All tests were performed on a Micron Pentium 133 machine with 64M of RAM and 100M of free disk space.

#### 4.2.1. Increase in drawing speed using Visio 5.0 over 4.x (x is 0 or 1).

In several years we have seen five changes in the version of Visio: 3.0 → 4.0 → 4.1 → 4.5 → 5.0. This rate of evolution seems typical of the shrink-wrapped component package market, and would not be surprising for smaller scale components based on component technologies such as COM. We used the exploration harness to compare the performance of versions 4.x and 5.0. Specifically, in order to evaluate whether the value added in switching from version 4.x to 5.0 was likely to be great, we wanted to know if version 5.0 could draw and iterate over shape objects significantly faster than version 4.x. Understanding performance was also important to our ability to represent the advantages and shortcomings of Galileo to our clients. The data that we obtained offered the following insights:

- There are insignificant differences between version 4.1 and 4.2 performance.
- The performance differences between versions 4.x and 5.0 are significant: As Figure 3 shows, the time to draw objects in version 4.x increases quadratically in the number of shapes already on the page, but in version 5.0 increases roughly linearly. This order-of-magnitude improvement was not clearly documented.

- In a previous test, we determined that the maximum number of shapes Visio 4.0 could store on a single drawing page was approximately 5400. Interestingly, version 5.0 retained this limitation.

![Figure 3. Comparison of times to draw shapes using Visio 4.1 and 5.0](image)

Understanding such curves provided an improved basis for design decision making. We decided to adopt Visio 5.0 as the “official” drawing component of our tool, for example.

#### 4.2.2. Increased drawing capacity of Visio.

Early versions of Galileo suffered from a limitation of Visio which restricted the number of references to shape objects that a client can hold to about 700. Galileo held references to shapes to manipulate the information they displayed, such as a name. As a result, fault-trees could contain no more than 700 logic gates. The Visio Corporation suggested that rather than retaining references (which required Visio to keep shape objects in memory) that we hold Visio shape unique identifiers that Visio can map to shape references dynamically. We used the exploration harness to understand how this design change would affect drawing performance. We determined that drawing performance was considerably slower using the numerical identifiers, but we had little choice, since we were required to be able to draw large trees. However, this knowledge drove us to try to find other ways to improve drawing performance. The answer turned out to be related to our use of screen updating, which is discussed in the following section.

#### 4.2.3. Using screen updating in Visio.

We expected drawing speed to increase when “screen updating” was disabled. Visio’s API provided a function to turn it on and off. Yet, we observed no performance benefit in early versions of Galileo, as revealed in measurements taken with our exploration harness. This information led us to investigate. We discovered that opening a document in Visio turned screen updating on as a side effect. We had...
been calling the functions in the wrong order. When we opened the document and then turned off screen updating, we saw the expected performance improvement. Through experimentation we learned more about the semantics of key functions of the component, and were able to improve our design. Figure 4 compares drawing performance with screen updating turned on and off.

Figure 4. Comparison of times to draw shapes with and without screen updating (Visio 5.0)

4.2.4. Potential performance improvements using alternate clients. The choice between using VBA and VSL clients as wrappers presented interesting opportunities to improve the performance of Word and Visio. Because VSL clients are dynamically linked into the Visio process, calls to Visio functions avoid inter-process calls. Figure 5 shows a moderate improvement in the VBA client’s ability over the EXE client to iterate over word objects in Word 97. Figure 6 shows an improvement in Visio’s ability to iterate over shape objects using a VSL as opposed to an EXE client. Although none of these clients have been implemented in Galileo, the exploration harness revealed information that suggest opportunities for future value-enhancing improvements.

Figure 5. Comparison of times to iterate over words using EXE and VBA Clients (Word 97)

Figure 6. Comparison of times to iterate over Visio shape objects using EXE and VSL clients

4.3 Outcomes

Our experience suggests that the following properties of our prototype are particularly valuable:

- **Interactive exploration**: The exploration harness offers a simple, efficient environment for creating exploration cases and executing them under widely varying conditions. In this respect, the tool is useful for learning how best to use a component.

- **Automated collection and presentation of data**: We employed package-oriented component-based design in building our harness by using Microsoft Excel 97 to display test data and to automatically generate graphs of those data. The graphs appearing in this paper were generated automatically using this component.

- **Component-based architecture**: It is important that changes in test cases do not require recompiling the entire exploration harness. The separation of the exploration harness from the test components alleviates this problem and makes it easier for third parties to create custom test components.

Our prototype exploration harness has some shortcomings. For example, we have not implemented a mechanism for saving selected sets of exploration cases. This is a drawback when performing what amount to regression tests on a newer version of a component. Second, creating the exploration components is too difficult. We separated the exploration components from the exploration harness but we find it cumbersome to create the exploration components. Writing them required learning the basics of COM and DLL’s, for example. Ideally, third parties who write exploration tests should not have to worry about these details. One possibility for
improvement is to use Microsoft’s Visual Basic programming language to create test components. Visual Basic hides many of the complexities involved in creating COM components, and may provide a simpler environment for writing test components.

Finally, two inherent limitations bear mentioning. First, getting the right answers depends on asking the right questions. That is, an exploration harness produces valuable information only after the designer has figured out what issues are important. The approach does not itself help the designer to focus on the right issues. Exploration harnesses are not a general solution to the problem that designers sometimes discover problems only when it is too late. Second, there are many critical questions that cannot be ascertained effectively by experimentation. Nevertheless, systematic, repeatable exploration of component properties under controlled conditions promises to be a valuable way to generate information about key component properties.

5. Related work

The exploration harness concept is not entirely new, of course. There is important related work in a number of areas. First, several organizations are doing research on lifecycle issues for component-based development. The Software Engineering Institute, for example, defines four stages in the component-based software development lifecycle: (1) Component Qualification: Choose a set of components based on the requirements of the system; (2) Component Adaptation: Augment components to conform to a particular architectural style; (3) Integration: Write custom code to combine the components into a working system; and (4) System Evolution: Update system to incorporate newer versions of components. Our work is closely related to the qualification stage. However, in the SEI model, qualification focuses only on the question of whether to use a given component or not. Our interactive, tool-assisted discovery approach is much more geared to acquiring detailed knowledge of components that have already been adopted.

Second, Garlan et al. describe compositionality problems in constructing a system from components. A key problem that they identified was that architectural properties of components that are important at the system level are often unknown and undocumented, only to be discovered too late. Garlan et al. focus on architectural issues and solution techniques, such as better documentation and bridging technologies. Our work is meant to focus directly on an issue that is even more fundamental to design: namely the designer’s inherent state of incomplete knowledge. Our approach, tool-support for interactive discovery, is novel in that regard.

Third, a great deal of work has been done over many decades on benchmarking, which involves the discovery of particular properties of given artifacts. Our approach is of course closely related. Our novel contribution is support for engineer-guided, interactive, tool-assisted discovery. We have also gained insight into some basic issues relevant to designing exploration tools. For example, we have employed a package-based style to ease the difficulty of collecting and presenting collected data.

6. Conclusions

The effort that we expended in creating an exploration harness and in collecting data about the packages used in the POP-based development of Galileo paid off in a number of ways. First, it offered clear recommendations on design changes needed to optimize certain performance-critical aspects of Galileo; for example, Visio’s drawing speed and capacity. Second, it offered empirical measurements of effects of design changes on performance. Such tests could be made before incorporating the changes into Galileo; for example, the effect of using screen updating in Visio. Third, it offered indications of the alterations in performance resulting from the evolutionary paths of commercial components: subsequent versions of the major components could be tested against older versions to determine possible improvements or design changes. Fourth, it offered repeated, systematic, iterative exploration of packages under controlled and varied conditions. Fifth, it offered an improved ability for us to do back-of-the-envelope estimates of performance and limitations of Galileo.

Iterated, repeatable, systematic exploration of component properties promises to play an important supporting role in CBSD. Furthermore, given that the important details of components will continue to remain unavailable, exploration moves closer to the center of system design. It appears that those most adept at ascertaining relevant component properties will do best in making decisions about whether and how to use components in systems. Tool support in the form of exploration harnesses appears to offer considerable promise in automating such exploration for competitive advantage.

7. Acknowledgements

This work was supported by NSF Grants CCR-9502029 and CCR-9506779, including supplementary funding for Research Experiences for Undergraduates (REU).

8. References


