

Software Components with Retrospectors

Chang Liu

Information & Computer Science
University of California, Irvine
Irvine, CA 92697-3425
USA
+1 (949) 824-2703
liu@ics.uci.edu

Debra Richardson

Information & Computer Science
University of California, Irvine
Irvine, CA 92697-3425
USA
+1 (949) 824-7353
djr@ics.uci.edu

ABSTRACT

Using COTS components in the software development process introduces new problems to software testing. In particular, a consumer (user) of a COTS component typically does not have source code access nor is she aware of how well the component has been tested or how to best test it embedded in her own application. She also doesn't have any well-established guidelines for integration test adequacy. This paper proposes a solution to this problem: software components with retrospectors, which record testing and execution history of a component and make testing information available to software testers. This paper analyzes the problems of testing component-based software, introduces retrospector techniques and discusses how retrospectors can help test component-based software.

Keywords

Software testing, software component model, COTS

1 INTRODUCTION

In 1987, Brooks speculated that software reuse (buy versus build) might provide a silver bullet for software engineers [Bro87], the hope being that software reuse would achieve higher software productivity and greater quality. Yet more than ten years later, the software engineering discipline is still not seeing satisfactory results. The recent emergence of component-based software models, however, gives the discipline new hopes. Widely used commercial component models, such as Sun's JavaBeans [Ham97] and Microsoft's COM [Rog97], establish consistent and powerful component-based software architecture models, which enable software developers to share their software very effectively.

With the help of these component models, software organizations and software developers are now able to produce software components and sell them to other organizations or developers who use them as building blocks in constructing their own software systems. Component producers typically don't have specific knowledge of the future software systems that will use these components or how they will be used. On the other hand, component consumers who use these components to

build software systems typically don't have access to the components' source code nor do they necessarily need to understand it. If the proper information is passed both ways, there could be a COTS (Commercial-Of-The-Shelf) software component market that would greatly benefit both component producers and component consumers.

This would be a wonderful world for software engineers. Yet, as the COTS component market emerges, several problems other than the architecture technology itself arise. Among these problems is software testing, which is the theme of this paper.

2 PROBLEMS OF TESTING COMPONENT-BASED SOFTWARE

Traditional software testing techniques usually assume that software testers have complete knowledge of software requirements, software implementation (source code), and the execution context (environment). In the world of COTS component-based software development, however, this is not the case.

First, COTS component consumers don't necessarily have access to the source code of these COTS components. The testers of component-based software, i.e. software systems that use COTS components, can't use existing integration testing techniques, which typically require access to source code and won't work in this situation. For example, the Path-Based Integration testing technique [Jor95] and object-oriented integration testing techniques [JE94], both need source code of components to construct Method/Message Path Graph to perform the testing. How can a consuming organization's testers perform integration testing and make sure the resulting software systems work properly without access to COTS component source code? They can't; thus, new testing techniques are needed.

Second, even if consumers do have access to all the source code of COTS components (for instance, by paying more to purchase them), they still need to test the component-based software thoroughly yet avoid repeating the producer's testing activity. After all, it is the producer's responsibility to fully test a COTS component before delivering it. The consumer should just have to make sure that their usage of the component is consistent with

its intended usage, that it doesn't emphasize any problems and that the whole software system is working properly. An additional problem observed by [Ros97], however, is that often software that uses a component might bypass a part of that component. Traditional test coverage techniques don't take this into account by supporting the skipping of unused source code. So, even with full access to COTS component source code, the component-based software tester equipped only with traditional testing techniques will have difficulty carrying out effective testing.

Third, COTS component producers do not completely know the future execution context of the component that they are writing. They usually make some assumptions. But without proper technique and tool support, COTS component producers can't express their assumption about the component explicitly. Mismatched assumptions of different components cause architectural mismatch [GAO95] for COTS component users who wish to integrate these components into one system.

Current popular component models such as JavaBeans and COM don't address these software-testing issues. As the COTS component market matures and adoption of COTS components increases, these testing problems will become a major barrier towards wider acceptance of these component models. In this paper, we propose a way to enhance component models and solve these problems by adding retrospectors to software components.

3 SOLUTION: ADD RETROSPECTORS TO SOFTWARE COMPONENTS

What is needed to solve the aforementioned problems of component-based software testing, is a mechanism to enable component producers to prepare the COTS components for any future usage, and to advise future component-based software developers about the testing of any component-based software containing these components. With the help of such a mechanism, testers of component-based software can perform testing efficiently, in particular, to test the component-based software thoroughly yet avoid repeating testing for tested COTS components. We propose a mechanism called retrospector that is added to each software components.

3.1 Definition of Retrospectors

A *retrospector* is a software entity that assists the testing of a software component by recording the testing and execution history of a component. A retrospector is analogous to an *introspector*, which is a JavaBeans fragment that records static component information [Ham97]. A retrospector communicates with builder tools and tester tools to make all related testing information available to software testers. Here, a *tester tool* is a

software tool with which software testers test COTS components and component-based software. It is analogous to a *builder tool*, a component for JavaBeans, which refers to a software tool with which software developers manipulate COTS components to build component-based software [Ham97]. A retrospector also records a component producer's recommendations about test cases for and usage patterns of that component. In short, while an introspector of a component records and reports static information of the component to builder tools, a retrospector records and reports dynamic information of the component to both builder tools and tester tools.

A *retro-component* is a software component with a retrospector. A retro-component is a software component in that it complies with all current requisites on a software component, no matter which component model is adopted. In addition, a retro-component has a retrospector that maintains testing and/or dynamic execution history.

3.2 Benefits of Retro-components

Retro-components can help with software testing for both component producers and component consumers who develop component-based software.

For component producers, retrospectors can help record testing history and perform coverage analysis. Furthermore, with the help of retrospectors, it is possible to collect execution information of a software component from component users. So retrospectors can also help carry out perpetual testing [Ost96].

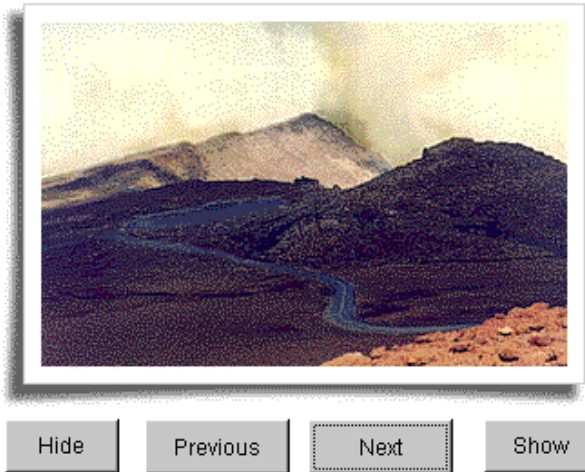
For component consumers, retrospectors can make a great difference. With the recommendation of test cases from retrospectors, component-based software testers can design test cases more easily and effectively. With the results of test coverage analysis from retrospectors, testers now have better knowledge of how extensive the tests have been thus far and can decide when to stop software testing more precisely.

Furthermore, retrospectors can assist component-based software developers other than testers. The usage patterns stored in retrospectors are actually precious design information of the components. They can help component-based software developers to understand COTS components better and quicker.

3.3 SlideViewer: an example

As an example, let's consider a component *SlideViewer*. Object *SlideViewer* is the only object inside this component. It has four methods:¹

- Previous: Go to the slide show's previous image
- Next: Go to the slide show's next image
- Hide: Hide the slide show
- Show: Show the slide show



Picture 1: Component *SlideViewer*

If the coding style of *SlideViewer* complies with a component model specification, for example, JavaBeans, *SlideViewer* becomes a JavaBeans component, a bean. To become a retro-component, component *SlideViewer* must also have a retropector. However, the component producer doesn't have to write a retropector, although it is recommended. If the coding style of *SlideViewer* complies with *Retro-Spec* (the specification common properties of retro-components), a default retropector can be created for this component. So a programmer can take advantage of all benefits of retro-components without doing any coding. The content of its default retropector would be as shown in Table 1:

¹ Of course, to be practical, *SlideViewer* would need more methods. For simplicity, let's suppose here that only these four methods are of interest.

As stated in [Szy98], a component is a unit of deployment and might contain more than one object. Again, for simplicity, let's focus on the simple case of one object, one component in this example.

ENTRY		VALUE
Supported coverage analysis methods		<i>Path coverage</i> <i>Statement coverage</i>
Path coverage	Definition	<i>Source code level path coverage</i>
	Recommended next test case	""
	Current coverage	0%
	recommended coverage	80%
	Test case generator	<i>Next untested meaningful test case</i>
Statement coverage	Definition	<i>Source code level statement coverage</i>
	Recommended next test case	""
	Current coverage	0%
	recommended coverage	95%
	Test case generator	<i>Next untested meaningful test case</i>
Meaningful test cases		["Show" "Next" "Previous" "Hide"]*
Coverage analysis method as recommended by component producer		<i>Statement coverage</i>
Execution history		<i>NIL</i>

Table 1: Content of default retropector of retro-component *SlideViewer*

Notice that the two default coverage criteria here are both source code based. This doesn't conflict with the goal of retropectors, to help testing without component source code. A retropector will include source code coverage analysis by providing a compiled version of the source code instrumented as necessary to support the coverage criteria. In this way, the users of retro-component don't have access to source code, but still can have access to the source code coverage analysis methods in the retropector.

Here in the default retropector, the "Meaningful test cases" expression `< ["Show" | "Next" | "Previous" | "Hide"] * >` was generated according to the four methods. This regular expression actually includes all possible combinations of any invocations of these four methods in any order.

The programmer can also customize the retropector. For example, she might add one user-defined coverage criterion to the retropector. In this example, we add a coverage

criterion that only cares about four user-defined test cases. After testers execute all of these four test cases, this user-defined coverage criterion will report 100% coverage. Even though very simple, this customized retrospector can greatly help unfamiliar users of this component to choose the first few test cases.

The programmer can also write his own "Meaningful test cases" expression to replace the default one. Usually the one that is written by the programmer will include more semantic information of method invocations and thus is more precise.

The resulting customized retrospector is shown in Table 2.

Besides a component producer, a component consumer, such as a tester, can customize the retrospector, too. For example, a tester can add one entry "Meaningful test cases specified by tester" and write an expression for his own interested test cases. He can then instruct the test case generator to use this expression instead of the one defined by component producer.

For example, if a tester of component-based software which uses component *SlideViewer* is not interested in button "previous", he can add his own "Meaningful test cases specified by tester" entry with a value <"Show" ["Next"] * "Hide">. Then the test case generator will not generate any test case related to "previous". This feature of retrospector is especially useful when COTS component consumers only use a fraction of functions provided by COTS components, which happens very frequently.

After this retro-component was executed twice in a tester tool, once with a test case that contains invocation to four methods in order of "Show", "Next", "Hide", once with another test case of "Show", the status of its retrospector will change as shown in Table 3.

Notice that here the parameters are not recorded in "Execution history". This is because in this example, none of the methods actually takes parameters. In other cases where parameters are significant, the instrumented code would capture their values and they would be included in "Execution history".

All information listed in the above tables are accessible to developers from tester tools or builder tools. Thus, with the help of retrospectors, testers of component-based software can very easily check the current status of any retro-component at any time.

ENTRY		VALUE
Supported coverage analysis methods		<i>User-defined coverage</i> <i>Path coverage</i> <i>Statement coverage</i>
User-defined coverage	Definition	<i>percentage of specified test cases executed</i>
	Recommended next test case	"Show"
	Current coverage	0%
	recommended coverage	100%
	Test case generator	<i>Next untested test case in the specified test case list</i>
	Specified test case list	"Show" "Show", "Hide" "Show", "Next", "Hide" "Show", "Next", "Next", "Hide"
Path coverage	Definition	<i>Source code level path coverage</i>
	Recommended next test case	"Show", "Hide"
	Current coverage	0%
	recommended coverage	80%
	Test case generator	<i>Next untested meaningful test case</i>
Statement coverage	Definition	<i>Source code level statement coverage</i>
	Recommended next test case	"Show", "Hide"
	Current coverage	0%
	recommended coverage	95%
	Test case generator	<i>Next untested meaningful test case</i>
Meaningful test cases		"Show" ["Next" "Previous"] * "Hide"
Coverage analysis method as recommended by component producer		<i>User-defined coverage</i>
Execution history		NIL

Table 2: Content of customized retrospector of retro-component *SlideViewer*

ENTRY		VALUE
Supported coverage analysis methods		<i>User-defined coverage</i> <i>Path coverage</i> <i>Statement coverage</i>
User-defined coverage	Definition	<i>percentage of specified test cases executed</i>
	Recommended next test case	<i>"Show", "Hide"</i>
	Current coverage	<i>50%</i>
	Recommended coverage	<i>100%</i>
	Test case generator	<i>Next untested test case in the specified test case list</i>
	Specified test case list	<i>"Show"</i> <i>"Show", "Hide"</i> <i>"Show", "Next", "Hide"</i> <i>"Show", "Next", "Next", "Hide"</i>
Path coverage	Recommended next test case	<i>"Show", "Hide"</i>
	Current coverage	<i>30% (not real data)</i>
	Recommended coverage	<i>80%</i>
	Test case generator	<i>Next untested meaningful test case specified by testers</i>
Statement coverage	Recommended next test case	<i>"Show", "Hide"</i>
	Current coverage	<i>60% (not real data)</i>
	Recommended coverage	<i>95%</i>
	Test case generator	<i>Next untested meaningful test case</i>
Meaningful test cases		<i>"Show" ["Next" / "Previous"] * "Hide"</i>
Meaningful test cases specified by tester		<i>"Show" ["Next"] * "Hide"</i>
Coverage analysis method as recommended by component producer		<i>User-defined coverage</i>
Execution history		<i>"Show", "Next", "Hide"</i> <i>"Show"</i>

Table 3: Content of customized retrospector of retro-component *SlideViewer* after execution of two test cases

3.4 Design principles

When specifying Retro-Spec, we want to follow the following principles, which are similar to the successful design principles of JavaBeans [Eng97].

3.4.1 Automatic Retrospection

The default behaviour of retrospectors should allow automatic retrospection of any retro-component. Developers can choose to implement a *RetrospectorInfo* class to provide testing information with its associated retro-component explicitly, or just to use predefined *RetrospectorInfo* class without doing anything. The predefined *RetrospectorInfo* class should become part of Retro-Spec.

As long as developers follow certain simple coding style, the default behaviour of retrospector should do a very satisfactory job. The default of automatic retrospection help keep retro-components simple to create and easy to use.

3.4.2 Design-Time vs. Test-Time vs. Run-Time

In Retro-Spec, a retro-component will have three modes, design-time, test-time and run-time. Instances of *RetrospectorInfo* class usually are used only in test-time mode and wouldn't be carried into run-time mode (this capability might, however, be desired for operational testing or perpetual testing). This can help keep retro-components lightweight in final execution mode.

3.4.3 No Requirement on Visibility

There is no requirement that a retro-component should be visible at run-time - e.g. at the user interface level. An invisible run-time retro-component should also be shown visually by tester tools. The retrospector technique works for any reusable software components, visible or not. However, currently most available COTS components are run-time visible components. So our experiment of retro-components will begin with visible components.

3.4.4 Flexible Test-Time Component Test Case Generators

Retro-Spec allows producers and consumer testers to associate custom test case generators with a retro-component. This custom test case generator, if written by component producers, actually records component producers' knowledge about the function and implementation of the retro-component and is very valuable during test time (both during continued testing by the producer or later testing by the consumer). On the other hand, being able to write their own custom test case generators, consumer testers can make use of it as a convenient testing tool.

3.5 Prototype Support

Our initial prototype support for Retro-Spec will be based on one popular component model, which we currently expect to be JavaBeans. However, we do not intend our Retro-Spec to be tied to any single component model. Instead, our goal is to keep the specification for retrospectors model-independent so that any component model could easily adopt it.

After choosing a component model as our base component model, we have two ways of implementing Retro-Spec. One way is to incorporate it as part of the component architecture specification and make it part of future versions of the component model specification. Retro-Spec doesn't require any additional language mechanism. It is a coding convention plus a few class libraries and a few tools (possibly including one pre-processor to instrument source code for source code coverage criteria). So, from a technical point of view, it's natural to make it part of the component model specification. However, as popular component model specifications such as JavaBeans and COM are maintained by industry companies and we're university researchers, there might be some barriers to this approach.

Another approach is to implement the Retro-Spec as a separate model on top of the component model specification. We will develop a separate set of tools to support retro-components. Although this approach will require greater effort, it seems the safer one to embark upon. It is always possible to adopt the second approach, and then after demonstrating success, to follow through with first approach.

4 RETROSPECTORS AND PERPETUAL TESTING

Researchers in perpetual software testing are working to support seamless, perpetual analysis and testing of software from requirement specification through deployment [Ost96]. If the users and developers of component-based software agree to keep retrospectors the final release version instead of deleting all retrospectors before software release, retrospectors can remain active after software deployment and keep collecting the actual usage, real interactions with real software users. The information collected by retrospectors can be sent back to software developers periodically under the agreement between software developers and users. This feedback is very valuable to software developers and could become the starting point of in-depth perpetual software testing practice.

Although perpetual testing is a very new research area, researchers have already found out two characteristics of perpetual testing. Perpetual testing should be continuous and incremental [Ost96]. As retrospectors collect testing

and execution information case by case, they can serve very well as a mechanism to support perpetual testing.

5 SUMMARY

Our research in component-based software testing is still in its infancy. The concepts described here are still in an experimental stage. Yet, the importance of this research is undoubtedly clear. As component-based software development becomes the main stream approach used by software engineers world-wide, testing component-based software will soon become the most time- & money-consuming part of software development, even worse than the impact on productivity and costs that software testing poses in today's software development paradigms.

We believe the retrospector technique will provide a very promising solution to component-based software testing problems. To store testing information inside a component is a convenient and effective mechanism that enables component-based software testers to test component-based software without access to the COTS components' source code.

REFERENCES

- [Bro87] F. Brooks, No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10-19, April 1987
- [Eng97] R. Englander, Developing Java Beans. O'Reilly, 1997
- [GAO95] D. Garlan, R. Allen, J. Ockerbloom, Architectural Mismatch: Why Reuse Is So Hard, *IEEE Software*, November 1995
- [Ham97] G. Hamilton (Editor), JavaBeans Specification, Sun Microsystems, <http://www.sun.com/beans>, July 24, 1997
- [JE94] P. Jorgensen and C. Erickson, Object-Oriented Integration Testing, *CACM*, September 1994
- [Jor95] P. Jorgensen, Software Testing: A Craftsman's Approach, CRC Press, 1995
- [Ost96] L. Osterweil, Perpetually Testing Software, The Ninth International Software Quality Week (QW'96), San Francisco, May 21-24, 1996
- [Rog97] D. Rogerson, Inside COM, Microsoft Press, 1997
- [Ros97] D. Rosenblum, Adequate Testing of Component-Based Software. Technical Report, UCI-ICS-97-34, University of California, Irvine. August, 1997
- [Szy98] C. Szyperski, Component Software - Beyond Object-Oriented Programming, Addison-Wesley, 1998