An Approach for Understanding and Testing Third Party Software Components

Jennifer M. Haddox * Cigital Labs * Dulles

Gregory M. Kapfhammer * Allegheny College * Meadville

Christoph C. Michael * Cigital Labs * Dulles

Key Words: Commercial-off-the-shelf (COTS) software, Software wrapping, Software reliability, Software reuse, Java, bytecode instrumentation, Fault Injection, Assertions.

SUMMARY & CONCLUSIONS

In this paper we present an approach to mitigating software risk by understanding and testing third party, or commercial-off-the-shelf (COTS), software components. Our approach, based on the notion of *software wrapping,* gives system integrators an improved understanding of how a COTS component behaves within a particular system. Our approach to wrapping allows the data flowing into and out of the component at the public interface level to be intercepted. Using our wrapping approach, developers can apply testing techniques such as fault injection, data collection and assertion checking to components whose source code is unavailable.

We have created a methodology for using software wrapping in conjunction with data collection, fault injection, and assertion checking to test the interaction between a component and the rest of the application. The methodology seeks to identify locations in the program where the system's interaction with COTS components could be problematic. Furthermore, we have developed a prototype that implements our methodology for Java applications. The goal of this process is to allow the developers to identify scenarios where the interaction between COTS software and the system could result in system failure. We believe that the technology we have developed is an important step towards easing the process of using COTS components in the building and maintenance of software systems.

1. INTRODUCTION

The use of commercial-off-the-shelf (COTS) software components has become increasingly prevalent in recent years. COTS components usage, however, has resulted in a new set of problems that are not present when large systems are built and maintained using *custom-built* software. Because a COTS component is not specifically created for the application into which it is integrated, it may not meet all of the application's requirements. Furthermore, the source code for a COTS component is rarely made available to the buyer. Even if source code were available, determining how a COTS component will behave once it is integrated into a software system can still be difficult indeed (Ref. 9).

In order to address these problems, we have created a methodology and a tool meant to aid developers when they attempt to integrate a COTS component into their applications. Our approach, based on the notion of *software wrapping*, focuses on helping system integrators gain an improved understanding of how a COTS component interacts with the rest of the system into which it is integrated. A key goal is to give system integrators a strategy for dealing with COTS components without becoming any more dependent on the vendor of the component than necessary.

The remainder of this paper is organized as follows. Section 2 discusses some issues regarding COTS component usage. Section 3 clarifies the motivation of our research and explains some of the goals it has intended to fulfill. Section 4 provides a high-level overview of software wrapping as well as a description of how it can be used in conjunction with other technologies to deal with some of the problems presented by COTS components. Section 5 describes our implementation of software wrapping. Section 6 deals with ideas for future work.

2. COTS COMPONENT OVERVIEW

The use of third party software components, such as commercial-off-the-shelf products, has become more and more common in the building and maintenance of large software systems. Corporate downsizing and decreased government budgets, as well as the spiraling costs of building and maintaining large software systems, have necessitated the reuse of existing software components (Ref. 16). In addition, reusing existing software can potentially reduce the time-to-market. Finally, another important issue to consider is that, simply put, programmers are expensive. Why pay an entire team of developers to create a component from the ground up, when it is cheaper to simply pay a few developers to integrate a pre-existing component into a new application? For reasons such as these, very few large-scale software systems are built from scratch. Commercial-off-the-shelf (COTS) components, legacy software, and custom-built components comprise today's large software systems (Ref. 6).

COTS components are defined by Vigder and Dean as “components which are bought from a third-party vendor and integrated into a system” (Ref. 13). However, according to Ref. 5, a more detailed and expanded view of COTS components should be taken. A COTS component could be as “small” as a routine that computes the square root of a number or as “large” as an entire library of functions. The important point is that a COTS component already exists and was created by people outside of the software development organization that will actually use it (Ref. 5).

Though employing COTS components in the building and maintenance of a large system can provide some benefits, COTS component usage presents some unique problems as given by Ref 1 and Ref. 13:

1. COTS component source code is often unavailable; thus, the component must be analyzed and tested as a “black box.”
2. Updates and evolution of a COTS component are provided by the vendor. New functionality of an updated component could be detrimental to specific applications that use it. In fact, functionality in the original component could also be problematic.
3. The vendor often fails to provide a correct or complete description of the COTS component's behavior. This can result in the buyer of the component having to guess how the component is meant to be used or how it is supposed to behave. Worse yet, the buyer could end up using the component in a manner the vendor did not intend. Unanticipated uses could compromise the reliability of both the COTS component and the application into which it is integrated.
4. Maintenance can become an issue because the vendor may not correct defects or add enhancements as the buyer needs them. Developers in the organization that purchased the component may be forced to make modifications themselves, which can be quite difficult if the component’s source code is unavailable or if the component’s specification is poor.

Clearly, the creation or maintenance of systems that use COTS components is by no means a trivial endeavor. On the contrary, integrating COTS components into an application is prone to error, can require a significant amount of coding, and can be problematic to test properly (Ref. 13).

3. MOTIVATION AND FOCUS OF THIS RESEARCH

Due to financial and time-to-market considerations, software development organizations have become increasingly reliant on software provided by third parties for functionality that is needed for the creation and maintenance of applications. The goal of our research was to create a tool and methodology that would return a greater measure of control to organizations using component-based systems. In creating our software wrapping technology, we address the following issues:

1. Since source code is often not available for a COTS component, our approach can be applied to a component regardless of whether the buyer has source code for the component. In short, our approach is not coupled to source code at all.
2. Though cooperation from the vendor is desirable, our approach can be applied regardless of the vendor's degree of involvement following the purchase of the component.

These issues provided a focus for the direction of our work. By keeping them in mind we developed the approach that is described in the next section.

4. USING SOFTWARE WRAPPING TO ADDRESS COTS CHALLENGES

System developers cannot depend on the vendor of the COTS component to ensure that the purchased component will behave properly in the system in question. Hence it is up to the system designers to ensure that the COTS component will not adversely affect the behavior of the system in which it will be used. In this section we present our version of software wrapping as well as a method for using it that will allow developers to mitigate COTS component risk.

4.1 High-level Overview of Our Software Wrapping Approach

The approach that we have researched and developed, which is based on the concept of *software wrapping,* can aid developers in their efforts to test and verify COTS components. Conceptually, this approach requires an additional layer of software, called a wrapper, to encase the COTS component. As shown in Figure 1, the wrapper is responsible for intercepting any input that the system might send to the component or output that the component would send back to the system. In the context of this research project, we define input and output to mean the information being given to and returned from the component at the public interface level. The hope is to isolate the COTS component during the testing process in order to understand whether or not it is interacting with the rest of the system as expected.

Once access to the component's input and output is gained, a variety of testing operations can be performed to provide an improved understanding of the component's interaction with the rest of the system. For example, the wrapper can be hooked up to a mechanism that simply monitors, records, and stores the input and output of the component for observation. This can be particularly helpful when the system is comprised of multiple COTS components. Consider the situation where one COTS component's output becomes another COTS component's input and source code is not available for either component. If both components are encased in wrappers, then the person testing the system gets a glimpse as to what is occurring when those two components interact.

In general, the user of a system that implements the wrapping approach can decide which testing techniques to use. For the purposes of our research, we focus on applying the testing techniques of fault injection, data collection and assertion checking to the input and output of a COTS component via wrapping. By fault injection (Ref. 14) we mean corrupting the data that is being passed between components to present the system with scenarios not found in typical testing. Data collection simply refers to exporting various internal data states of the system to a file, database, etc. for later observation (as described in the example in the previous paragraph). Assertions (Refs. 10, 11, 15) are rules for system behavior defined by the developer. In the context of this paper, these rules specify acceptable behavior at component interfaces. These rules are integrated into the application and checked at runtime. If a rule is violated, then some sort of error handling code executes to handle the situation, or at the very least, notification is given that an assertion has been violated.

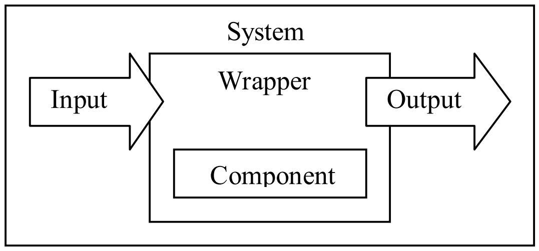

Figure 1: Conceptual View of a Wrapper

4.2 A Method for Applying Software Wrapping

Though it would be ideal to be able to gain a complete understanding of the internal implementation of an individual COTS component, it is still of value to learn about the data the COTS component will interchange with the buyer's system. Thus, understanding the public interface of the component and what the rest of the system expects when interacting with that interface is of benefit. This process can begin by the development organization establishing requirements for the system's interaction with the new component. Those building the system should have an idea of what types of inputs their system will provide to the COTS components and what types of information the system can expect to receive back from the component. Armed with this knowledge, the system developers can now begin to utilize the concept of wrapping. For it is the wrapping technique we have just described that will allow our methodology to be applied to a COTS-based system (or more generally, any system consisting of some components whose source code is not available).

The goal of this process is to allow the developers to identify scenarios where the interaction between COTS software and the system could result in system failure. The first step developers must take is defining assertions to govern how the COTS components in question should interact with the application of which they are a part. The application is then tested with the assertions in place. During these executions fault injection is applied at component interfaces in an attempt to determine under what situations the system (and the individual COTS components) can fail.

Specifically, we can use fault injection to supplement the input values that the COTS components would receive from the system. Using fault injection to perturb the components' "normal" inputs can allow for observation of how these components react to inputs not usually encountered during testing. In this case we are trying to determine which kinds of inputs make these components fail. We can then begin to refine the assertions that define acceptable input to the components based on the reaction of the components to perturbed input.

By perturbing the input, we can also see when the assertions that define the COTS components' acceptable output are violated. If certain types of input always result in such a violation, then we can further strengthen the assertions defining the input based on this information. The goal of this process it to identify scenarios that would make the COTS components crash, and then account for those situations in the assertions.

Fault injection can also be applied to the COTS components' output to determine how the rest of the system will react to data produced by these components. Once outputs that result in failure are identified, the assertions defining the components' output can be refined as well.

When an assertion in the system is violated by fault injection, or some bug existing in the system causes it to fail, data collection comes into play. Data collection can be used to export state information at locations in the code where fault injection is applied. Also, of use is collecting data where assertions fail. The developers can of course specify that data collection occur at specific locations where they would like more information about the internal state of the system. This can provide the developer with helpful information if the system fails on a particular test run. As problems with the code are discovered through this testing process, the tester may want to modify the application (if possible) to account for these problems. The methodology that we just described appears in Figure 2.

Consider the following example to clarify this discussion. Assume a group of developers has a component X that is to be integrated into their system. For a particular operation in X that takes an integer *i* as input , the developers create an assertion specifying that whenever the system invokes this operation, *i* must be greater than zero. The developers then specify that data collection be performed whenever this particular operation in X is invoked. When the developers test the entire system and apply fault injection to *i*, they may notice that X does indeed cause a system failure when *i* is less than zero. They may also note that X causes a failure when *i* is assigned values greater than one thousand. Thus, they may want to redefine their assertion to state that *i* must be greater than zero *and* less than one thousand.

By creating assertions and applying fault injection, and then studying the additional output of the program provided via data collection, the developers can gain an improved understanding of how the COTS components are interacting with the application. They can then refine their assertions to account for potential failure situations they discovered by stepping through the process we have discussed. The end goal is to create assertions that will account for as many situations as possible that can result in system failure.

Once these potential problem scenarios are discovered and assertions are created to account for them, some action must be taken for the case of assertion failure. With our current implementation of the methodology, the only option is for the developers to modify their own code to account for problems that could result from the COTS software being integrated into the system. In some cases this is appropriate. A COTS component could very well be the source of problems because some *custom built* component in the system may be using it improperly. Thus, the custom component *should* be modified in this case.

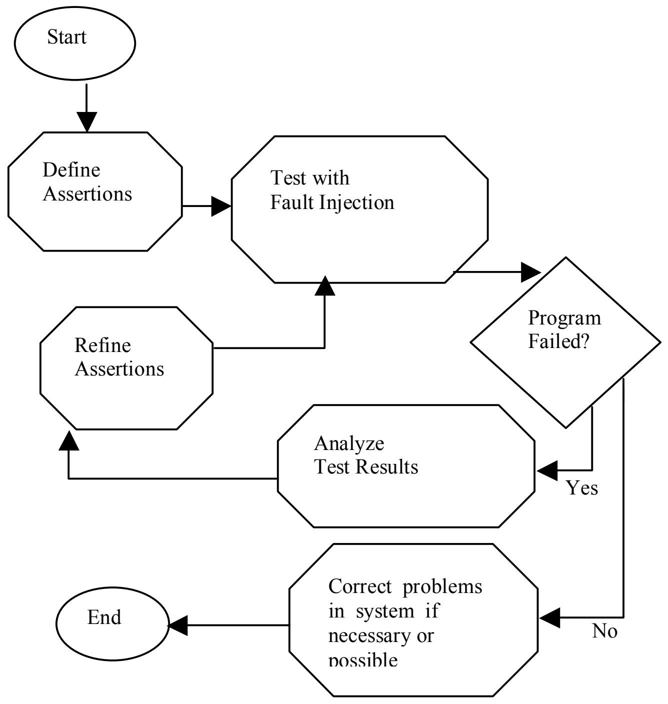

Figure 2: Overview of Methodology

It is certainly possible, however, that the COTS component itself is the problem. In this case, it would be ideal for the developers to have some method of modifying that component's behavior. As stated, the wrapping technology we have developed does not allow functionality to be modified in a component whose source code is not available; it only allows input and output data to be intercepted. Thus, the developers must adapt their system to account for problems caused by COTS software. In Section 6 we describe an extension of our current prototype that would allow for COTS functionality to be modified. First though, we give a description of the tool that we built.

5. IMPLEMENTING SOFTWARE WRAPPING

Now that we have explained the high-level overview of our approach and how it can be applied to a COTS-based system, we shall describe the underlying technology that makes it possible. This section will provide a brief listing of the requirements for our tool, a description of our implementation, and a summary of the benefits and drawbacks of this implementation.

5.1 Requirements

Several groups, such as Ref. 3 and Ref. 9, have already focused on some variation of the concept of wrapping. Our research has specifically focused on devising a method to apply wrapping to object-oriented components whose source code may not be available. We initially chose Java as our candidate language due to its increasing popularity. From this point on, we shall define a *component* as a *Java class,* and the two terms will be used interchangeably.

When performing the analysis to devise a solution for wrapping object-oriented components, we kept the following general requirements for a wrapping prototype in mind:

1. Wrapping a component must not change the core functionality of any program that utilizes the wrapped component. While a wrapping approach might introduce a slight execution overhead, it should not destroy the functionality of systems that previously executed in a correct fashion
2. Our implementation must allow the methodology described in Section 4 to be applied to Java applications.
3. The user of a wrapping tool should be able to turn the wrapping mechanism on and off at system start-up time. For example, a system integrator should be able to add a "wrapper flag" to the command line statement that starts an application in order to indicate that wrappers should become active.
4. The wrapper must be able to handle any operation that is advertised by a Java interface and provide the ability to intercept information at method entries and method exits (whether it is an expected or unexpected exit), thus allowing input and output data to be intercepted as described in Section 4.

5.2 Implementation

We have taken the general notion of software wrapping and built a prototype that allows this approach to be applied to object-oriented components written in Java. Before describing this approach, we shall provide some basic information about Java. All programs written in the Java programming language are converted into bytecode by the Java compiler. The bytecode is then interpreted and executed by the Java Virtual Machine, which is a sort of "abstract computer" that can be implemented in the hardware but normally is implemented in the form of a software program (Ref. 4). Thus, bytecode is basically the portable assembly language of the Java Virtual Machine (Ref. 4).

The approach that we employed ended up being somewhat unconventional, in that it goes against the conceptual view of a wrapper given in Section Four. The solution is specific to Java and relies on bytecode instrumentation. The original component's bytecode is instrumented such that the input and output of a method can be captured and monitored.

Using our bytecode instrumentation technology, one can access and modify the bytecode of a Java class that is to be wrapped. More specifically, one can intercept and capture the input and output of methods of the class to be wrapped and transfer that information to some subsystem that would perform testing operations on the captured input/output data. Our technology uses a third party software package called JavaClass (Ref. 2) to aid in the bytecode instrumentation process.

Additionally, we have implemented the testing techniques described in Section Four. We have created fault injection, data collection and assertion checking subsystems that can be used in conjunction with our wrapping tool. More information on these subsystems and how they are used in conjunction with our implementation of wrapping can be found in Ref. 7.

Note the fact that all instrumentation of a class to be wrapped is performed at runtime. This is accomplished via a *custom class loader*, which intercepts the request to load a class to be wrapped. All classes are loaded from disk into the Java Virtual Machine at runtime by a class loader that is part of the Java Development Kit (JDK) (Ref. 12). However, it is possible to create one's own class loader that can be used in place of the one that comes with the JDK. We took this approach in order to allow our tool to instrument the bytecode of a class at runtime. Before a class to be wrapped is loaded into the Java Virtual Machine, our bytecode instrumentation subsystem gains access to it and modifies it. Then, the instrumented version of the class is loaded into the Java Virtual Machine. Once the execution of the application completes, no trace of the modifications is left behind. The original class still resides on disk.

The user creates an Instrumentation Configuration File (ICF) in which the classes to be wrapped are specified. One can also use an ICF to specify that only certain methods in a class be wrapped (i.e., certain methods are instrumented such that their input/output is intercepted). This file then becomes input to our system, and based on what is specified in the file, certain classes are intercepted by our custom class loader, instrumented, and then loaded into the Java Virtual Machine by our custom class loader. The rest of the system never has knowledge that wrapping occurred in the first place. See Figure 3 for a high level diagram of the wrapping subsystem.

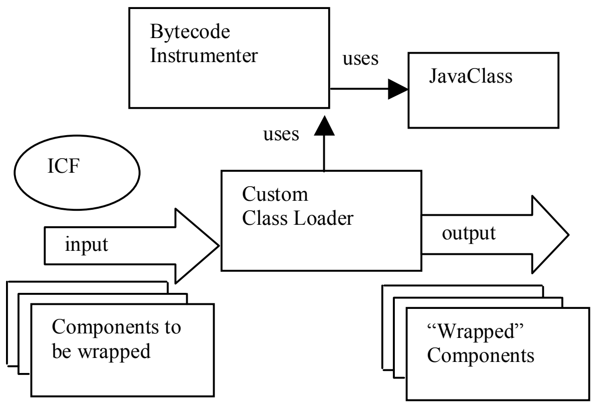

Figure 3: High Level Overview of Wrapping System

5.3 Benefits to Our Approach

There are several important advantages to our approach that are worth noting. First of all, the approach is in no way dependent on access to the source code of the component being wrapped. Further, it is not dependent upon support of the vendor of the component. In keeping with our requirements, the wrapping mechanism can be turned "on" or "off" at the time the system in question is to be executed and does not corrupt the normal behavior of that system. Additionally, our approach allows testing techniques such as fault injection and assertion checking to easily be applied to a component whose source code is unavailable. Thus, our implementation supports the methodology described in Section 4. Finally, a user can easily create his or her own subsystem that would perform some testing operation on the captured input and output. Our system was designed such that a new subsystem could easily be used in conjunction with our wrapping technology.

5.4 Drawbacks

Though this approach is effective, it does suffer from some drawbacks. A potential limitation with this approach has to do with the fact that all methods in Java have a 64KB limit on their size. Since we are increasing the size of a method with this approach as we are adding to its bytecode, it could be possible to exceed this limit, though it is not likely. Additionally, one cannot use direct bytecode instrumentation to wrap native methods (functions written in some other language, such as C, that can be invoked by a Java program) since native methods have no bytecode. Thus, our solution does not completely fulfill the requirement that a wrapping system should be able to handle any operation advertised by a Java interface. Finally, our actual implementation is coupled to Java as it relies on bytecode instrumentation.

6. FUTURE WORK

6.1 Continuing Evaluation

In the future we would like the opportunity to test our prototype and methodology on a real world Java application that was partially constructed from third party components. We did complete some preliminary testing on our tool by using a testbed system that we constructed for this project. When applying our tool to the testbed system, the tool always produced "wrapped" classes capable of running within the Java Virtual Machine and interacting with the other classes without causing a disruption in the expected behavior of the system. Using the wrapping system in conjunction with the data collection, fault injection and assertion checking subsystem, we were able to uncover errors in the testbed system (Ref. 7). Using our tool on a real world Java application, however, would provide a much more rigorous test of the effectiveness of our approach.

6.2 Extending the Technology

We eventually hope to use the research we conducted and the prototype we developed to create a methodology, loosely based on the methodology presented in Ref. 8, and a tool whose primary purpose is the disabling or modification of problematic functionality in COTS components. At a high level, our methodology would consist of three steps:

1. Learn the behavior of the component within the system.
2. Modify the appropriate portion of a component with problematic functionality.
3. Test the modified component in the system.

The user of this tool will be given several methods of learning about the behavior of the component in question. The user will be able to apply our fault injection, data collection, and assertion checking techniques, all of which we will expand and improve for the construction of this tool. Additionally, techniques for machine-learning based inference of component behavior will be developed. We will also create a unit-testing framework that can be used to test the component in isolation. We will then build on our bytecode instrumentation technology in order to allow the user a means of modifying a component's functionality. The user will then be able to utilize our fault injection and assertion checking systems to test the modified component to determine if the changes made are having a negative impact on the system's behavior.

The unit-testing framework would be used to perform regression testing on the component. Before the component is modified, our data collection system can be used to record and store inputs to the component during execution of the system. The unit-testing framework will be capable of feeding these stored inputs back into the component. The goal will be to determine if the behavior of the modified component is different from that of the original.

We believe that the creation of such a tool could further aid those using COTS components in the building and maintenance of their applications.

7. CONCLUSIONS

The increasing use of third-party COTS components can in theory lead to reduced costs and faster development cycles, but these advantages can come at a steep price. With their applications dependent on the behavior of components from third parties, developers and integrators have suffered a loss of control. Thus, our goal with this project has been to help developers and integrators regain some control of their COTS-based systems.

With this research project, we have developed an innovative approach to testing object-oriented COTS components. We have created a methodology based on software wrapping meant to provide developers with an improved understanding of a COTS component's behavior within a system. We have created an implementation of this methodology for Java applications. In the future we look forward to improving and extending our tool and methodology and applying it to real world applications.

8. ACKNOWLEDGMENTS

We would like to thank Dr. Timothy Tsai, Michael Schatz, Richard Mills, Ryan Colyer, Richard Leslie, and Glenn Buckholz for the work they did on the research project described in this paper. We also thank Dr. Anup Ghosh for providing comments and feedback on this paper as it was being written. The work presented in this paper was funded by Army Research Labs under contract DAAD17-99-C-0052.

REFERENCES

1. C. Braun, "A lifecyle process for the effective reuse of commercial-off-the-shelf (COTS) Software," *Proceedings of the fifth Symposium on Software Reusability,* 1999.

2. M. Dahm. "Byte code engineering." JIT, 1999.

3. S. Edwards, B. Weide, J. Hollingsworth, "A framework for detecting interface violations in component-based software," *IEEE Computer Society Proceedings 5th International Conference on Software Reuse* Victoria, Canada, Jun. 1998.

4. D. Flannagan, *Java in a Nutshell,* 1999; O'Reilley and Associates, Inc.

5. W. M. Gentleman, "Effective use of COTS (commercial-off-the-shelf) software components in long lived systems," *Proceedings of the 19th International Conference on Software Engineering,* 1997.

6. A. K. Ghosh, J. Voas, "Inoculating software for survivability," *Communications of the ACM,* Jul. 1999, 42, 7.

7. J.M. Haddox, G. M. Kapfhammer, C.C. Michael, M. A. Schatz, "Testing commercial-off-the-shelf components with software wrappers," *Proceedings of the 18th International Conference and Exposition on Testing Computer Software*, 2001.

8. G. M. Kapfhammer, C.C. Michael, J.M. Haddox, and R.A. Colyer, "Identifying and understanding problematic COTS components," *Proceedings of the International Software Assurance Certification Conference*, 2000.

9. J.C. Knight, R. W. Lubinsky, J. McHugh, and K.J. Sullivan, "Architectural approaches to information survivability," Technical Report" CS-97-27, University of Virginia, Sept. 1997.

10. D. Mandrioli, B. Meyer, *Advances in Object-Oriented Software Engineering*; 1992; Prentice Hall.

11. D. Rosenblum, "Toward a method of programming with assertions," *Proceedings of the 11th International Conference on Software Engineering*, 1992.

12. B. Vennners, *Inside the Java 2 Virtual Machine,1999;* McGraw-Hill.

13. M. Vigder, J. Dean, "An architectural approach to building systems from COTS software components," Technical Report 40221, National Research Council, 1997.

14. J. Voas, G. McGraw, *Software Fault Injection: Inoculating Programs Against Error, 1998;* John Wiley and Sons.

15. J. Voas, K. Miller, "Putting assertions in their place," *Proceedings of the International, Symposium on Software Reliability Engineering*, November 6-9, 1994.

16. J. Voas, J. Payne, R. Mills, J. McManus, "Software testability: An experiment in measuring simulation reusability," *Proceedings of ACM Sigsoft,* Seattle, WA., April 29-30, 1995.

BIOGRAPHIES

Jennifer M. Haddox
Cigital Labs
21351 Ridgetop Circle, Suite 400
Dulles, VA 20166 USA

Internet (e-mail): jhaddox@cigital.com

Ms. Jennifer M. Haddox is a Research Associate at Cigital Labs. In this position, Ms. Haddox has served as the technical lead of the Component Reverification project, sponsored by Army Research Labs. In this role, she oversaw the design and implementation of a system that generates wrappers for Java classes using runtime bytecode instrumentation. Prior to joining Cigital full-time, she completed two internships with the company. During her time at Cigital, Ms. Haddox has had the opportunity to gain experience on such topics as fault injection, data collection, assertions, bytecode instrumentation, and object-oriented analysis and design. She has also co-authored several research papers and proposals. Ms. Haddox graduated summa cum laude from Allegheny College in May 2000 with a B.S. in Computer Science.

Gregory M. Kapfhammer
Allegheny College
Box Q 520 North Main St.
Meadville, PA 16335 USA

Internet (e-mail): gkapfham@allegheny.edu

Mr. Gregory M. Kapfhammer is an Instructor in the Department of Computer Science at Allegheny College. He is responsible for the development and teaching of the classes in the Department's new Applied Computer Science major. Mr. Kapfhammer focuses on the teaching of advanced undergraduate courses in the fields of software engineering, software testing, computer security, and distributed systems. Presently, Gregory is also a graduate student in the Computer Science Department at the University of Pittsburgh. Mr. Kapfhammer's research interests include software component wrapping, efficient regression testing, and the testing and analysis of distributed systems. Mr. Kapfhammer is involved in the development of several research prototypes that either utilize or extend the Java programming language and the Jini network technology. During past employment, Gregory was a Research Associate at Cigital Research Labs.

Christoph C. Michael
Cigital Labs
21351 Ridgetop Circle, Suite 400
Dulles, VA 20166 USA

Internet (e-mail): ccmich@cigital.com

Dr. Christoph Michael is a Senior Research Scientist at Cigital Labs. As a graduate student, he designed and implemented the fault-injection analysis algorithms used in Cigital's commercial product, the WhiteBox Software Analysis Toolkit. Dr. Michael has authored or co-authored 23 publications on software testing, test-data generation and intrusion detection in information systems. He has served as principal investigator on software assurance grants from NIST's Advanced Technology Program and the Army Research Labs, as well as software security grants from DARPA. His current research interests include information system intrusion detection, software test data generation, and dynamic software behavior modeling. A member of IEEE and INNS, Dr. Michael received a B.A. in Physics from Carleton College and an M.Sc. and Ph.D. in Computer Science from The College of William and Mary.