We need new ways to validate software components, specifically those deployed in diverse software environments. We must also consider the likelihood of realizing anticipated savings and whether component-based systems can meet reliability and availability requirements.

Testing Component-Based Software: A Cautionary Tale

Elaine J. Weyuker, AT&T Labs

Developing large industrial software systems with very high reliability and availability requirements entails enormous costs. Many organizations have begun to consider implementing such systems using reusable component repositories, with the expectation that

- commercial off-the-shelf (COTS) components or components designed for reuse can significantly lower development costs and shorten development cycles; and
- using them will ultimately lead to software systems that require less time to specify, design, test, and maintain, yet satisfy the high reliability requirements.

Many organizations see the arrival of languages designed to implement software components as the silver bullet they have awaited for decades. They envision that writing a component once and reusing it in many subsequent software systems will effectively amortize the development cost among all users. They assume they can seamlessly integrate repository components into a new environment, and that COTS components are plug-and-play. In either case they see these components as building blocks that can be easily incorporated into a software system to provide specified functionality.

To realize the potential benefits of such an architecture, however, we must design systems using reliable components that interoperate safely. This in turn means developing ways to test the components that make up a system, both in isolation and once integrated.

The Ariane 5 Lesson

In June 1996, during the maiden voyage of the Ariane 5 launch vehicle, the launcher veered off course and exploded less than one minute after takeoff. The Inquiry Board convened by the ESA's director general and the CNES's chairman determined that the explosion resulted from insufficiently tested software reused from the Ariane 4 launcher. Developers had reused certain Ariane 4 software components in the Ariane 5 system without substantial retesting, having assumed there were no significant differences in these portions of the two systems.[1]

Jean-Marc Jézéquel and Bertrand Meyer[2] argued that a reuse specification error caused the Ariane 5 failure and that had the design-by-contract approach been used, with its reliance on Eiffel-like assertions, the fault almost certainly would have been caught and the failure prevented. Ken Garlington[3] responded that they had demonstrated neither the necessity nor the sufficiency of using such assertions to prevent the Ariane 5 explosion or a similar disaster.

Although both sides offered cogent arguments, one thing is clear: as the reuse of software components and the use of COTS components become routine, we need testing approaches that will effectively guard against the occurrence of similar disasters. In addition, these testing techniques must be widely applicable regardless of the source code's availability, because whether we're using legacy code, purchasing off-the-shelf components, or accessing in-house component repositories, typically only the object code is available. We also must ensure—in a cost-effective way—that the resulting systems are sufficiently reliable to meet their specified needs.

Testing Newly Developed Software

Some commercially developed software systems go through little or no systematic testing, which can lead to serious consequences once the software has been released to the field. The later in the life-cycle that software faults are identified, the greater the cost of repair and the more serious the impact on the end user.[4]

Software systems should go through at least three stages of correctness testing:

- unit testing, in which individual components are tested;
- integration testing, in which the subsystems formed by integrating the individually tested components are tested as an entity; and
- system testing, in which the system formed from the tested subsystems is tested as an entity.

Many industrial software systems, including those my company produces, may also go through feature testing, performance testing, load testing, stability testing, stress testing, and reliability testing. These are distinguished by the granularity of the software entities being tested and the ultimate goal of validation.

The Ariane 5 disaster showed that not testing components in their new context may have disastrous consequences.

Unit testing frequently uses test cases selected using the component's actual source code, in which case we call it program-based or white-box testing. Unit testing is generally done by developers who have access to the source code and are familiar with its details, and therefore can constructively use this information. Also, the relatively small size of the individual modules or units being tested makes it feasible to consider the code details when determining appropriate unit test cases.

In contrast, system testing typically uses test cases selected without reference to the code details, because at this level, there is generally far too much code to rely on such details. We call a testing strategy that does not rely on code details black-box or specification-based testing. People other than the code developers usually do system testing; they may therefore be unfamiliar with the level of detail necessary to perform code-based testing and generally do not have access to the source code. They are only responsible for testing the fully integrated system; when they find symptoms of faults (that is, when failures occur in response to test cases), they simply transmit the information to the development organization for fault isolation and repair.

Either the developers themselves or an independent test organization may perform integration testing. The testers therefore may or may not rely on code details to select test cases, depending on the size of the subsystems tested and who performs the testing. Integration testing frequently emphasizes the interface code since the individual modules being integrated have already been tested.

Load testing occurs after the system's functionality has been thoroughly tested. This phase determines whether the system's resource allocation mechanisms function correctly and how the system behaves under particular loads. In my organization, it frequently includes stability, stress, reliability, and performance testing.

Alberto Avritzer and I[5] presented load testing algorithms that rely on the software's operational profile or operational distribution definition to select a meaningful and manageable set of test cases. An operational profile is a probability distribution that describes how the software will be used when operating in the field. This strategy emphasizes those software states most likely to be entered once the software is operational. We designed the test case generation algorithm to test very large industrial software systems with enormous state spaces (or input domains) that make it impossible to test even a small fraction of possible cases. The cases selected for testing are those most likely to occur and therefore most likely to impact the user if a fault associated with the state exists.

Developers reusing a component need to do considerable testing to ensure the software behaves properly in its new environment.

In some cases, it may prove essential to stress not only the most common situations but also the most critical or costly ones. For that reason, recent work[6] proposed incorporating the consequence or cost of failures into the test case selection process.

Testing Component-Based Software

David Garlan, Robert Allen, and John Ockerbloom[7] presented a case study detailing the problems that arose as they built a system from reusable components. Performance problems resulted from both the system's massive size and its complexity, which was frequently inappropriate for the tasks performed. The authors reported that they had trouble fitting selected components together; in some cases it took significant reengineering to make them interoperate properly. Maintaining the synthesized systems also proved difficult in the absence of low-level understanding.

These problems all impact a component-based system's quality and reliability, but this lack of low-level understanding has particular implications. Among them is the difficulty of developing test suites for such systems, both because necessary insights are not available and because lack of access to needed artifacts prevents certain types of testing.

Reusing components developed for a different project

When developing a component for a particular project or application, with no expectation of reuse, testing proceeds as usual. Information about the software's intended or expected usage usually affects the selection of test cases. Even with no explicitly defined operational distribution, testers usually have some information or intuition about how the software will be used and therefore emphasize, at least informally, testing of what they believe to be its central or critical portions.

These priorities will likely change, however, if it is decided to incorporate the component into a different software system. The original system may commonly execute portions that the other never will, which makes much of the testing irrelevant to the new user in the new setting. Likewise, the new user may use parts of the component that correspond to extremely unlikely scenarios in the original system's behavior, and these may have been untested or only lightly tested when the software was developed.

Changing priorities between applications implies that a software component developed for one project should not be used for another project without significant additional testing. The Ariane 5 disaster showed that not testing components in their new context risks significant and potentially catastrophic failures. This holds true whether the software was custom-designed in-house for a particular project and is being reused by a different project, was purchased as a COTS component, or comes from an in-house repository. While the techniques used to test components under these different circumstances will likely differ, it remains necessary to test components in their new environments.

As an example, let's examine two projects with significantly different usage patterns that use the same software component. I created a test suite using one of Avritzer and Weyuker's automatic test case generation algorithms.[5] This approach models the software as a Markov chain and uses operational distribution data to select test cases that exercise the states most likely to be executed in production.

Table 1 shows the first system's operational distribution data; Table 2 shows this data for the second system. The specification requires that each system handle a maximum of 1,000 simultaneous inputs (telephone calls), and that it handle three distinct types of calls. This implies 167,668,501 distinct Markov states for each system, hence that number of potential test cases—clearly a prohibitive amount of testing.

For Project 1, we created a test suite consisting of 88 test cases. This suite exercised only 0.00005 percent or roughly 1 in 2,000,000 of the system's potential states, but covered more than 98 percent of the probability mass associated with Project 1's operational distribution. When applied to Project 2, however, the same suite covered less than 24 percent of the probability mass associated with the operational distribution. Hence very few of Project 2's critical states were tested at all, and we had little evidence that the software would function properly in its intended environment.

Clearly, then, developers reusing the component in Project 2 would need to do considerable additional testing to be sure the software behaves properly in its new environment. However, Project 2 developers might not have access to all the original artifacts—the source code, detailed requirements, or specification documents—nor would they typically have the low-level, detailed understanding of the system architecture that only the component's developers would likely have. This could make it difficult or impossible to adequately test the component when integrating it into Project 2, which could have potentially disastrous results—even if the time, personnel, and will existed to do the testing.

During traditional system testing, the testers do not have access to the source code because they do not need it, not because it is not available—the development organization normally has the source code at that stage. When testing a component-based system, however, the source code and other artifacts may be entirely unavailable. In the ideal case, the component was carefully specified, all documentation was retained, the source code is available, and test plans and test suites have been maintained. System testing must be redone for the new environment but can proceed as for other projects. Even in this case, however, debugging might be significantly more difficult because the development team may no longer be available or, even if they are, may no longer be familiar with the component's code. The development team might also feel that debugging and modifying the component is not their responsibility, leaving those tasks to other developers who are not particularly familiar with the code.

When the component was written at some other time for some other purpose, and only the object code is available in a repository, testers face another significant problem: if they detect evidence of faults, how can these be isolated and corrected if the source code is not available?

Table 1
Project 1 Call Traffic Data

Call Type	Average Arrival Rate (calls/min)	Average Holding Time (min)
r1	3.0	1.0
r2	1.0	1.0
r3	0.3	1.0

Table 2
Project 2 Call Traffic Data

Call Type	Average Arrival Rate (calls/min)	Average Holding Time (min)
r1	0.3	1.0
r2	1.0	1.0
r3	3.0	1.0

Using components developed for multiple users

Software components developed explicitly to be used in many different environments fall into two categories: those created for an in-house component repository and COTS components.

Initial testing of repository components cannot depend on operational distribution or intended usage patterns, since there are likely to be many end users, each with different usage patterns. The development testers must therefore try to envision the component's many possible uses and develop a comprehensive range of test scenarios. Because the development testers cannot envision all potential scenarios, prudent users will do a significant amount of testing when integrating the component into their system.

Thus, even code written just once will eventually need more testing, which could offset much of the savings of component reuse. Without source code and access to the personnel and expertise used to create the component, testing and debugging prove significantly more difficult—and therefore expensive—than in a traditional development environment. It may also be impossible to perform adequate levels of validation, which could lead to low system quality. Any code modifications necessary for the component to operate properly in the new environment further reduce possible savings. Worse, modification may be essentially impossible without the source code unless reverse engineering is done, typically at great expense.

COTS components present many of the same challenges. These components have gotten a lot of press recently, leading many organizations to consider them as an answer to the the lack of experienced developers and the enormous cost of building large systems. Some see it as the only way to get a competitive advantage by being first or early to market. But every quality or reliability problem outlined above is magnified when integrating off-the-shelf components into a system.

Without source code and access to the personnel and expertise used to create the component, testing and debugging prove much more difficult—and therefore expensive.

The first and most obvious problem is the almost certain lack of source code, precluding any modifications for either debugging or extensions. The second problem is the certain lack of detailed knowledge that Garlan and colleagues[7] found so problematic. Also, users will likely have little or no control over the component's maintenance and support. How frequently will it be updated? A COTS component may also have considerable functionality that your system will never need; this extra code could cause possible interoperability problems and negatively impact your system's performance.

Perhaps the single most important quality issue concerns what might happen if the vendor decides to cease supporting the component, or goes out of business entirely. How will a project dependent on the component survive? Some have proposed creating escrow agreements for the source code in such cases, but even if the vendor agrees to that, will the potential problems I've cited make it infeasible to maintain the component at any reasonable level of cost and reliability? The bottom line is this: If the project has very high reliability and availability requirements, your reputation is at stake. Telling your customers it was someone else's fault that the product is unavailable or behaves unreliably will not do them much good, nor will it relieve your responsibility.

Facilitating Reusable Component Testing

If one purpose of designing component-based software is to reuse the components, then we must give significant thought to repository design so that components will be available for multiple projects. Obviously, we must decide how to store the components and identify them for ready access when new systems are being designed. But that will not suffice—we must also associate, with each component, a person or team primarily responsible for maintaining and, in some cases, distributing it. This person or team should also serve as a resource for projects using the component to build new software systems or enhance an existing system's functionality by adding new features. In addition, for the component to be truly reusable, we must ensure that interface standards exist to which developers strictly adhere.

Several requirements and testing activities can decrease potential problems associated with component reuse and thereby improve the resulting software's quality. For each software component built, several related artifacts should be stored and appropriately updated when the component is modified. These include the following:

♦ The software specification, including pointers between individual requirements and their implementations. When either the specification or the code is modified, this information makes it easy to track and complete associated modifications, thereby helping to guarantee that the specification remains up-to-date.

♦ The test suite, including pointers between individual test cases and those portions of the code they were designed to test. This makes it easy to keep track of the appropriate test cases that must be rerun for regression testing when the code is modified. When new functionality is added, new test cases should be added to the test suite. Keeping these pointers will highlight any functionality that has not yet been tested.

♦ Pointers between the specification and parts of the test suite designed to validate a particular functionality described in a given part of the specification. This shows which parts of the specification have been tested lightly, inadequately, or not at all. When parts of the specification change, it will be easy to identify those parts of the test suite that require augmentation or must be rerun.

In all cases, the test suite should include both inputs and expected outputs. This will facilitate regression-testing of changes to a component.

If organizations seek to develop truly useful component repositories, they must carefully consider component design, interfaces, associated directories, storage conventions, and maintenance arrangements. Components will likely need to be tested for each new environment so that developers and users can better predict their expected behavior and performance once installed. Only with this investment of time and resources will components become the company assets many people envision they can be. We must also think long and hard about whether reusing existing components or using COTS components can truly be cost-effective and provide the reliability assurances that many of today's industrial environments require. ❖

REFERENCES

1. J.L. Lions, "Ariane 5, Flight 501 Failure, Report by the Inquiry Board," http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html, 19 July 1996.
2. J.-M. Jézéquel and B. Meyer, "Design by Contract: The Lessons of Ariane," *Computer*, Jan. 1997, pp.129-130.
3. K. Garlington, "Critique of 'Design by Contract: The Lessons of Ariane,'" http://www.flash.net/~kennieg/ariane.html, Mar. 1998.
4. B. Boehm, *Software Engineering Economics*, Prentice Hall, Upper Saddle River, N.J., 1981.
5. A. Avritzer and E.J. Weyuker, "The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software," *IEEE Trans. on Software Eng.*, Sept. 1995, pp. 705-716.
6. E.J. Weyuker, "Using Failure Cost Information for Testing and Reliability Assessment," *ACM Trans. on Software Eng. and Methodology*, Vol. 5, No. 2, Apr. 1996, pp. 87-98.
7. D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch or Why it's Hard to Build Systems out of Existing Parts," *Proc. 17th Int'l Conf. Software Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., Apr. 1995, pp. 179-185.

About the Author

Elaine Weyuker is currently a technology leader in the Research Division of AT&T Labs, Florham Park, New Jersey. From 1977 to 1995, she was a professor of computer science at New York University's Courant Institute of Mathematical Sciences. Her research interests include software engineering, particularly software testing and reliability, and software metrics. She is also interested in the theory of computation, and coauthored (with Martin Davis and Ron Sigal) *Computability, Complexity, and Languages* (2nd ed.), published by Academic Press.

Weyuker received an MSE from the Moore School of Electrical Engineering, University of Pennsylvania, and a PhD in computer science from Rutgers University. She is an ACM fellow and a senior member of IEEE, is a member of the editorial boards of *ACM Transactions on Software Engineering and Methodology* and *The Empirical Software Engineering Journal*, and is an advisory editor for *Journal of Systems and Software*. She has served as the secretary/treasurer of ACM SIGSOFT, on the executive board of the IEEE Computer Society Technical Committee on Software Engineering, and as an ACM national lecturer.

Address questions about this article to Weyuker at AT&T Labs–Research, Room E237, 180 Park Ave., Florham Park, NJ 07932; weyuker@research.att.com.