Certifying Off-the-Shelf Software Components

Off-the-shelf components could save the software industry considerable time and money. However, the industry first needs a set of black-box processes to certify the suitability of COTS components.

Jeffrey M. Voas
Reliable Software Technologies Corp.

The cost and development time of software could be significantly reduced if only there were a widely used component industry. Even the best programmers can churn out only 10 lines of code per day. With systems like those used in cellular telephones—some consisting of 300,000 lines of code—custom software development can become very expensive.[1] If developers could purchase 100,000 lines of code, they could save 10,000 programmer-days, creating less expensive software and moving it to market more quickly.

The savings for developers could be staggering. If a world-class programmer costs $500 per day, purchasing 100,000 lines of code would result in saving $5 million (minus licensing fees). Companies producing components would also see great rewards. If $1 million were an acceptable licensing fee for 100,000 lines of code, then only five licenses would pay for that component, and the producer would profit on subsequent sales. Admittedly, this simple analysis ignores a number of variables, but it does show the potential for components in the software industry.

Of course, many other industries use components—for automobiles, radios, bridges, and so forth. Each of these products relies on interchangeable parts. The software industry is analogous to other industries because most software today is built from smaller software objects. The software industry differs, however, in that it lacks the ability to confidently swap components in and out of systems. Developers are not now sure if a replacement component is as reliable or more reliable—in terms of logical quality—than the replaced component. If reliability could be gauged, software component commerce would flourish, and designing and repairing systems would become less expensive.

Developers need to know two things about a component: whether the component itself is reliable and whether the system will tolerate the component. When a component's reliability cannot be determined, people often assume that more expensive components are more reliable because they have undergone more testing. But sometimes less expensive components are more reliable simply because they have seen more usage. However, even if the reliability of a component could be known, there is never any assurance that it will fit smoothly into the system and not cause problems.

Software components are often delivered in "black boxes" as executable objects whose licenses forbid decompilation back to source code. Often source code can be licensed, but the cost makes doing so prohibitive. We therefore have developed a methodology for determining the quality of off-the-shelf (OTS) components using a set of black-box analyses. This methodology will provide developers with information useful for choosing components and for defending themselves legally against someone else's imperfect OTS components.

FIVE SCENARIOS

When considering a candidate component, developers need to ask three key questions:

- Does component *C* fill the developer's needs?
- Is the quality of component *C* high enough?
- What impact will component *C* have on system S?

As shown in Table 1, the answers to these questions create five basic scenarios. They emphasize that the OTS quality problem is not just a matter of component quality, but also a matter of integration compatibility. Even a dozen highly reliable components combined together do not guarantee a highly reliable system.

Determining which scenario a component falls into is subjective. For example, the only difference between scenarios 1 and 3 is component quality. If a component has good but not perfect quality, which scenario would apply? (If this were easy to decide, there wouldn't be dozens of software reliability models that all give different results.) A good certification methodology,

0018-9162/98/$10.00 © 1998 IEEE

Table 1. Key questions concerning the use of component C in system S.

Scenario	Is C what is needed for S?	Is C of high enough quality?	Does C have a positive impact on S?
1	Yes	Yes	Yes
2	Yes	Yes	No
3	Yes	No	Yes
4	Yes	No	No
5	No	N/A	N/A

then, will determine which scenario a candidate component falls into.

Today there are various approaches to software certification. One popular approach is simply to require the developer to take oaths concerning the development standards and processes used. Our certification methodology, however, is not based on this honor system. We assume that only a description of what the component does is available and that buyers must themselves determine whether the component is what they want. Our scheme doesn't even use information provided by the component vendor on exhaustive OTS testing and verification. Our methodology is based on the belief that totally independent certification is the only safe approach for component buyers.

Figure 1 shows our methodology's basic steps. The first step is to decide whether the component has the functionality needed. Any component that does not meet the developer's needs can be ignored. The three pivotal decision points, however, are labeled A, B, and C. Decision A helps developers ask whether the component's quality is sufficient. Decisions B and C help developers ask whether the system can tolerate the component.

Our certification methodology uses automated technologies like black-box testing and fault injection to determine whether the component fits into scenario 1, 2, 3, or 4. A component in scenario 4 would be of poor quality and have a negative impact on the system. A component in scenario 3 would be of poor quality but would have a positive impact on the system (which contradicts the popular idea in software engineering that imperfect software is always bad). A component in scenario 2 would be like a rejected transplanted organ: It would have the right functionality and be of high quality, yet it would be unsuitable for the system. Finally, a component in scenario 1 would enjoy the best of all worlds, being of high quality and appropriate for the system.

Determining whether a candidate component belongs to scenario 5 simply requires finding out whether the specifications of the component match the requirements of the system. Although matching must be done manually, mistakes are unlikely if developers make a reasonable effort to assess what the component does and how it connects to its environment.

Once the methodology determines the correct scenario for a candidate component, the developer can decide whether to adopt it in its current form, modify the functionality of the system or the component, or simply find a new component.

CERTIFICATION TECHNIQUES

Our methodology uses three quality assessment techniques to determine the suitability of a candidate OTS component. *Black-box component testing* is used to determine whether the component is of high enough quality—a consideration of decision A in Figure 1. *System-level fault injection* is used to determine how well a system will tolerate a failing component. *Operational system testing* is used to determine how well the system will tolerate a properly functioning component. Even these components can create system wide problems. System-level fault injection and operational system testing are used for decisions B and C in Figure 1.

Black-box component testing

Black-box testing comprises software testing techniques that select test cases regardless of the software's syntax. Black-box testing requires an executable component, an input, and an oracle, which is a manual or automated technique that determines if failure has occurred by examining the output for each program input. In contrast, white-box testing considers the code when selecting test cases. Since source code will not be available to OTS component buyers, white-box testing techniques will be of little help.

Our methodology uses black-box testing based on the system's operational profile—its distribution of test cases when the software is put to use—to determine the quality of components that can execute on their own.[2] Black-box testing should be used even when the OTS supplier has already tested the component because it is never clear how rigorous that testing was and what generic profile was used.

Black-box testing, however, can fail to exercise significant portions of the code, which is worrisome to developers attempting to certify components. Also, the value of black-box testing depends on accurate oracles, because faulty ones can allow certification of bad software and prevent certification of good software.

Our certification methodology depends on accurate oracles, but we know that some component buyers will have problems finding them. We recommend that buyers develop their own oracle according to what they want the component to do. Buyers then can test the quality of the component against what it is required to do and not necessarily against the vendor's claims.

Black-box testing has already been used by B.P. Miller and associates in their Fuzz model.[3] Fuzz provides a low-resolution approximation of the robustness of particular Unix utilities like ls. Fuzz works by

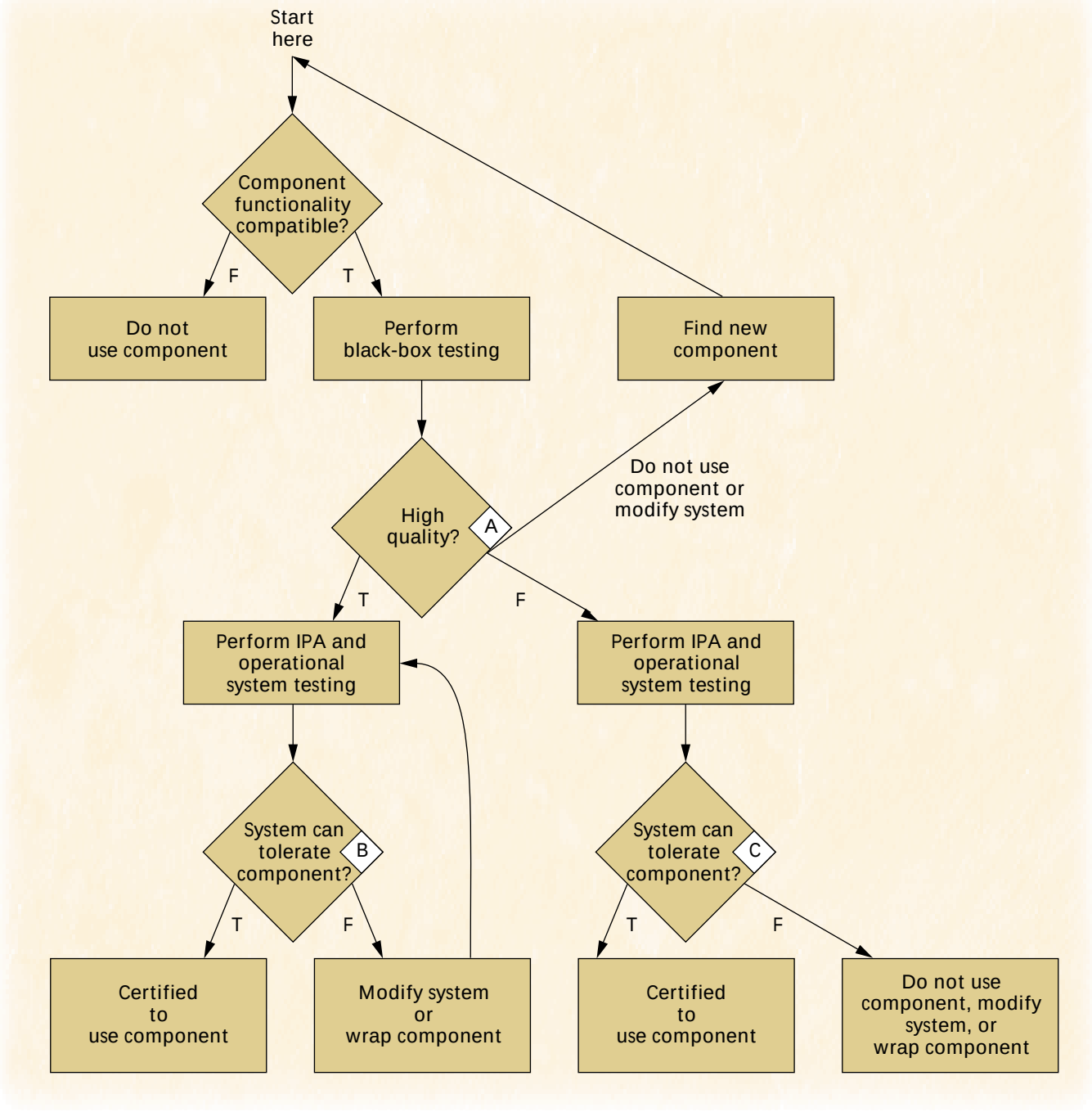

Figure 1. The OTS component certification process.

generating random black-box inputs, feeding them to Unix system functions, and watching for core dumps. The core dump, or crash, provides a crude way of determining whether the software has problems.

One serious problem with OTS software is that it can have unknown, malicious functionality, including, for example, Trojan horses. Only one test case might find it, and that test case probably won't be used during black-box testing. It is wrong, then, to assume that black-box testing will catch such serious problems. Black-box testing plays an important—though limited—role in assessing component quality.

The cost of black-box testing must also be considered. Buyers must generate inputs, create an oracle, and probably build a test driver, all of which are expensive tasks. However, these costs are minor compared to the potential cost and time savings of OTS components.

System-level fault injection

Even if a buyer knows that a component is free of Trojan horses and is of high stand-alone quality, the certification process is not yet complete. The next step is system-level fault injection, which tests systems by creating errors in them. Quality components can still result in unreliable systems, and poor quality components may cause no system problems; the system may simply ignore the faults. System-level fault injection

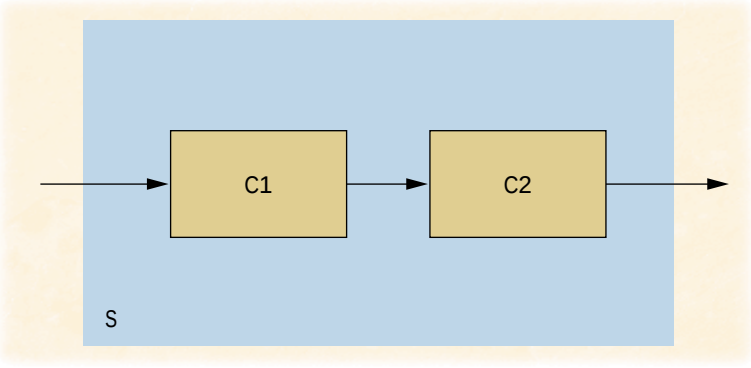

Figure 2. The system S made from components C1 and C2.

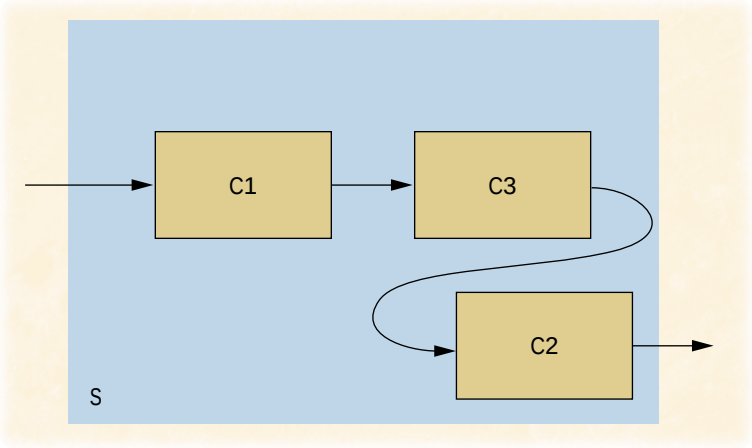

Figure 3. The system S augmented with OTS component C3.

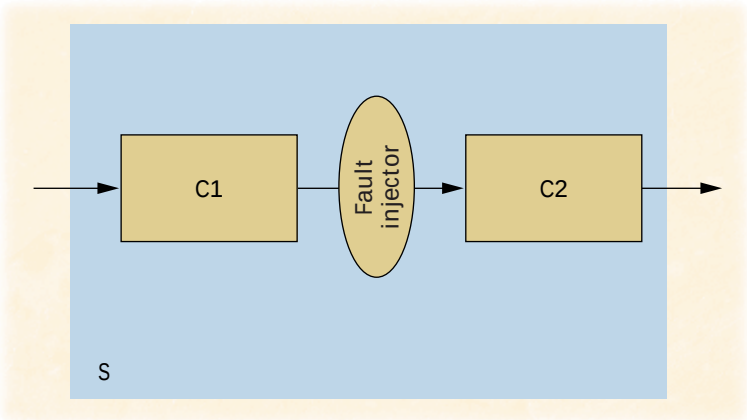

Figure 4. Simulating the failure of candidate component C3.

does not explicitly show how reliable the system is, but, rather, predicts how badly the system might behave if a component fails.

There are many types of fault injection, and the underlying ideas are not new, having been applied in hardware system validation, hardware design validation, and software testing.[4-7] Our certification methodology uses the fault-injection technique called *interface propagation analysis* (IPA).[8,9] IPA *perturbs*, or corrupts, the states propagated through interfaces between components. To perturb states, the buyer must have access to the interfaces that components use for communication. Our methodology uses a small software routine called a *perturbation function* to replace the original state with a corrupt state during software execution.

In our methodology, IPA (as well as operational system testing) determines the answer for the last column in Table 1: Does component C have a positive impact on system S? By corrupting data moving from a predecessor component to a successor component, IPA stimulates failure of the predecessor component. In this way, IPA determines whether the system can tolerate anomalies in a component.

To determine the impact of a component failure, IPA must know what component failure modes to inject and what system failure modes to look for. System failure modes include faulty system output data, faulty global system data, and corrupted data flowing between successive components. Since OTS component buyers are system builders, they should know what constitutes a system failure. System integrators, however, are less likely to know which component failure modes to simulate with IPA.

Therefore, our methodology uses an approach similar to input generation in Fuzz, but it uses a pseudo-random number generator to modify data in various, sometimes random, ways. If a system can tolerate totally random failures, it likely can tolerate real component failures. Our tools corrupt data and simulate component failures using generic, low-level fault injection algorithms.[8]

As an example of this analysis, Figure 2 shows system S composed of two components, C1 and C2. The output of C1 is the input of C2. Figure 3 shows a new version of S in which an OTS component C3 has been inserted between C1 and C2. In this example, the component buyer has already used black-box testing to determine the quality of C3 but now needs to know how the system will react to it. The buyer learns this through system-level fault injection, which will cause corrupt data to flow between C1 and C2, simulating a situation where C3 fails, as shown in Figure 4.

We have applied this analysis to systems relying on operating system utilities. One tool automates IPA for AIX operating systems and standard C library function calls. The key to applying fault injection to function calls or calls to the operating system is determining how the call returns information. The way a function is

declared defines what interfaces must be worked with. The output from an AIX function may be

- a return value,
- a changed value in an argument that is passed by reference to the AIX utility, or
- a combination of the two.

The buyer applies fault injection to the information returned to the calling application from the C library functions, simulating a failure of the entity that produced the information without worrying about how exactly the information got corrupted.

In AIX, `double cos(double x)` indicates that the `cos()` function receives a double integer (contained in variable x) and returns a double integer. This declaration defines the information flow into and out of the function. Because of C's language constraints, the only output from the `cos()` function is the returned value, and hence that is all that fault injection should corrupt. This declaration does not, however, specify what the function should do.

As mentioned earlier, IPA can be fine-tuned to simulate precise failure classes or simply random corruptions. For instance, based on common trigonometric mistakes, it would be reasonable to force the result of `cos()` to take on the values of NaN and Infinity. It would also be reasonable to employ 0, because this is a common incorrect result from cosine implementations. To see how this analysis is handled inside an application using `cos()`, consider the following:

```
if (cos(a) > THRESHOLD) {
     // do something
}
```

When perturbing the return value, the following instrumentation characterizes the process:

```
if (PERTURB(cos(a)) > THRESHOLD) {
     // do something
}
```

The buyer adds the fault injection instrumentation to the source application before the application is compiled. When the modified application is executed, a fault injection management process collects statistics on how tolerant the application was to forced corruptions.

Another example is the `strcpy()` function:

```
char *strcpy(char *s1, char *s2)
```

The `strcpy()` function copies the characters contained in string s2 into string s1. The value `strcpy()` returns is the pointer s1. To see how this OTS function is handled during fault injection, consider the following call:

```
strcpy(old, new)
```

For this function, corruption is performed upon exit of the original call:

```
strcpy(old, new)
old = PERTURB(old)
```

After the call to `strcpy()`, the result contained in `old` is replaced with a different string of the same size. As with the `cos()` function, we can take advantage of information about common faults occurring in string copy implementations (for example, by employing an empty string, which mimics incorrect handling of a dropout condition in the `strcpy()` logic). Since it is rare to ever use the value returned by this function, the pointer would typically not get corrupted. If, however, we did want to corrupt the pointer, we could do this:

```
PERTURB(strcpy(old, new))
```

In summary, system-level fault injection provides a means for assessing how well an application can recover after receiving bad results from OTS components. The beauty of this approach is that it concentrates on making the system more robust rather than on determining why returned information is bad.

One way to modify the component is through *wrapping*, or putting a software "wrapper" around it to limit what it can do.

Operational system testing

Operational system testing—which embeds an OTS component and executes the full system to determine how well the system will tolerate the component—complements system-level fault injection. Unlike fault injection, operational system testing does not employ functions to perturb states, because it executes original states without modifying output information.

An advantage of operational system testing is that the system actually experiences component failure, which provides a more accurate assessment of system tolerance. The downside is that an enormous amount of system-level testing is needed for components that rarely fail. Operational system testing, though, should be performed to ensure that a component is a good match for the system.

DEFENSE THROUGH WRAPPING

Even when testing shows that a system will not tolerate a component, a buyer may still decide to use the component, perhaps because it is all that is available. In this case, the buyer needs to make the best of a less-than-perfect situation—usually by modifying the system or the component. One way to modify the component is through *wrapping*, or putting a software "wrapper" around it to limit what it can do. Wrappers anticipate undesirable outputs and keep

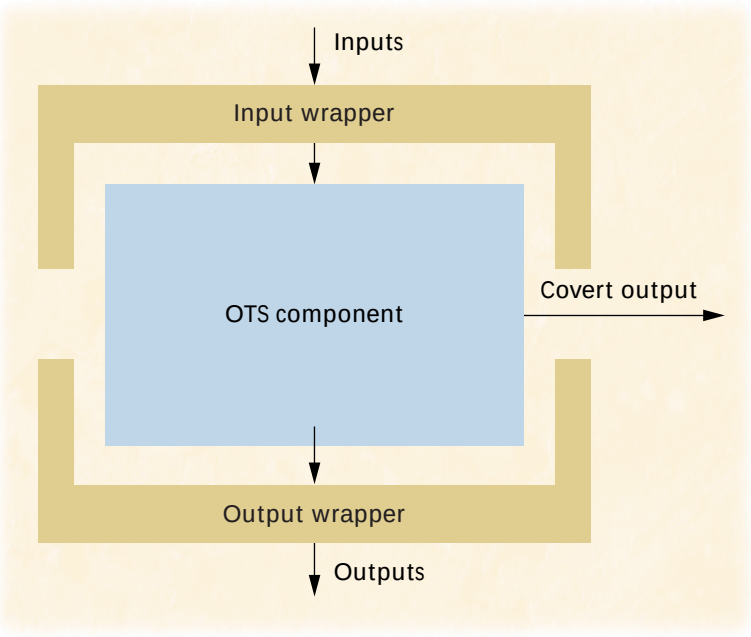

Figure 5. A wrapper failing to stop a covert channel.

them from occurring. Wrappers do not directly modify the component's source code (like removing lines of code) but instead indirectly modify and limit the component's functionality.

There are two types of wrappers. One keeps certain inputs from reaching the component, preventing it from executing on them and thus limiting the component's output range. Another type of wrapper captures the output before the component releases it, checking the output to ensure it meets certain constraints and passing it on if it qualifies. Both can be used at the same time.

Wrappers are not foolproof. As Figure 5 shows, illegal outputs can sometimes bypass a wrapper. For example, if it is not known that a component can call the operating system to delete a file, the wrapper probably won't be designed to prevent it. If, however, it is known that the component needs to delete temporary files that it creates, the wrapper can be designed to allow only file delete requests pertaining to those temporary files. The value of wrappers, then, depends on how well they are designed. They cannot protect against events unanticipated by developers—such as those events caused by Trojan horses.

Wrapper design can be aided by fault injection because it determines what outputs need to be protected against. By knowing what classes of component failure the system cannot tolerate, designers can create wrappers to filter outputs in those classes. Designers should also reapply black-box testing to the component to ensure that the installed wrapper is filtering correctly. This should be followed by another round of fault injection to confirm that the wrapper has improved system tolerance.

CERTIFICATION DECISIONS

If after applying black-box testing, IPA, operational system testing, and defense building, component C is in scenario 1 or 3, then it should be certified for use in system S. If it is in scenario 2, however, the decision is slightly more difficult because we know that component failures will be infrequent, but when they occur, S cannot tolerate them. Of course, any component in scenario 4 or 5 cannot be certified.

Most developers will choose to embed only OTS components that are in scenarios 1 or 3. For those components in scenario 2, they will either modify the system, find a better component, or accept the risk that the component will cause the system to fail.

Commerce in off-the-shelf software components is gradually becoming a reality. What is now keeping widespread adoption from occurring overnight is, first, not knowing the quality of components and, second, not knowing how systems will tolerate them. If these problems can be overcome, then developers will be better able to determine the opportunity costs of selecting one component over another or over creating custom software.

Our three-part methodology can help developers decide whether a component is right for their system. It is best performed during the evaluation period that most vendors provide. The process shows developers how much of someone else's mistakes they can tolerate. If they find that there are too many mistakes and the component won't work well in their system, the methodology provides several options they can take.

Our approach is not foolproof and perhaps not right for everyone. For example, our methodology does not certify a component for use in all systems. We have been conservative: Until we know how to assess components in a manner that accounts for all undesirable behaviors that could be forced into any system, it is prudent only to certify components for the idiosyncrasies of each system.

To foster an emerging software component marketplace, we believe that it must be clear for buyers whether a component's impact is positive or negative. Ideally, buyers would have this information before buying a component. Component buyers could then choose an appropriate component and apply static and dynamic techniques to determine its impact on the system. With this information, system builders could make better design decisions and be less fearful of liability concerns. And component vendors could expect a growing marketplace for their products. ❖

Acknowledgments

This work has been partially supported by DARPA Contract No. F30602-95-C-0282, Rome Laboratory Contract No. F30602-97-C-0117, NASA Contract No. NAS2-98052, and NIST Advanced Technology Program Cooperative Agreement No. 70NANB5H1160. I thank Frank Charron for his insights on AIX function calls and Lora Kassab for her suggestions on an earlier draft of this article.

References

1. S. Baker, G. McWilliams, and M. Kripalani, "Forget the Huddled Masses: Send Nerds," *Business Week*, July 11, 1997.
2. J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability Measurement Prediction Application*, McGraw-Hill, New York, 1987.
3. B.P. Miller, L. Fredrikson, and B. So, "An Empirical Study of the Reliability of Unix Utilities," *Comm. ACM*, Dec. 1990, pp. 32-44.
4. R.A. Demillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, Apr. 1978, pp. 34-41.
5. J.A. Solheim and J.H. Rowland, "An Empirical Study of Testing and Integration Strategies Using Artificial Software Systems," *IEEE Trans. Software Eng.*, Oct. 1993, pp. 941-949.
6. J.A. Clark and D.K. Pradhan, "Fault Injection: A Method for Validating Computer-System Dependability," *Computer*, June 1995, pp. 47-56.
7. J. Arlat et al., "Fault Injection for Dependability Validation: A Methodology and Some Applications," *IEEE Trans. Software Eng.*, Feb. 1990, pp. 166-182.
8. J. Voas et al., "Gluing Together Software Components: How Good Is Your Glue?" *Proc. Pacific Northwest Software Quality Conf.*, Pacific Northwest Software Quality Conf. Inc., Portland, Ore., 1996, pp. 338-349.
9. J. Voas, F. Charron, and K. Miller, "Robust Software Interfaces: Can COTS-based Systems Be Trusted Without Them?" *Proc. 15th Int'l Conf. Computer Safety, Reliability, and Security (SAFECOMP '96)*, Springer-Verlag, Berlin, 1996, pp. 126-135.

Jeffrey Voas is the guest editor of this issue. His biography appears on page 45.

Contact Voas at Reliable Software Technologies Corporation, 21515 Ridgetop Circle, Suite 250, Sterling, VA 20166; jmvoas@RSTcorp.com.