Software Component Interoperability

Jeffrey Voas
SAIC

When a software system fails, a confusing and complex liability problem ensues for all parties that have contributed software functionality (whether commercial off-the-shelf [COTS] or custom) to the system. This article explores the interoperability problems created by defective COTS software components, and, in particular, the hidden interfaces and non-functional behaviors. It also looks into the problem of composing non-functional behaviors that are related to quality-of-service attributes.

When there are complaints about the quality of the software, they are usually directed at the software's behavior, and not the underlying text [1]. Therefore it is important to define what constitutes bad behavior in software.

Bad software behavior is a function of the *environment* that the software resides in, and the environment is a mixture of the underlying hardware, the operating system, potential threats, the operational profile, other external software components that interact with the software, and the manner in which the software will be used [2]. Since these factors affect the software system's behavior, the architectural plan for how components are to be integrated should be, in part, based on these factors.

For example, a toaster comes with a warranty and that warranty assumes a particular environment, such as sitting on a kitchen countertop. Now take a working toaster, put it in another environment, like a bathtub full of water, and the behavior of the toaster is not going to be the same, and the warranty no longer applies. The same argument applies to software.

Will the Real Operational Profile Please Stand Up?

One of the most important issues during requirements generation is defining, as best as possible, the software environment; its operational profile is key.

Each piece of software has a set of input vectors that may be executed during testing or field usage. That is the software's *input domain*. The *operational profile* is simply a probability distribution function (PDF) for the input vectors of the input domain. This means each input vector has a certain probability of being chosen during testing or field usage, and the PDF is what defines the probabilities for each input vector. For example, if the input domain has 10 input vectors, and each input vector is as equally likely to be selected as any other vector, then each vector has a 10 percent chance of being selected. (Note that the operational profile is highly important when system-level reliability testing occurs [3].)

Thus, without an accurate description of the operational profile, predictions concerning how the software will behave in the field are unlikely to be accurate since each individual input vector can cause differing behaviors. Therefore, spending the extra time to contemplate the eventual operational profile and target environment is often as important as defining the software's functionality during requirements definition.

Tolerating Component and Subsystem Failures

More and more software is delivered to system integrators in black-box form; these components are packaged as executable objects (with licensing agreements that forbid decompilation back to source code), e.g., dynamic link libraries that originate from third-party sources, e.g., commercial off-the-shelf (COTS) software. A worthy goal, then, is to provide a methodology for determining how well a system can perform when particular black boxes are of such poor quality that interoperability and integration problems are almost inevitable.

One technique for assessing the level of interoperability between COTS software components and custom components is called Interface Propagation Analysis (IPA) [4]. IPA perturbs (i.e., corrupts) the states that propagate through the interfaces that connect COTS software components to other components. IPA is one form of *software fault injection* [5]. By corrupting data going from one component to a successor component, failure of the predecessor is simulated, and its impact on the successor can be assessed. This approach allows for measuring the level of intolerance when one component fails, sending junk information (or even a lack of information) to its neighbor.

To modify the information (states) that components use for inter-component communication, write access to those states is required (to modify the data in those states during the simulation). This is done by creating a small software routine called PERTURB that replaces the original output state with a different (corrupted) state. This is, of course, done as the system executes. By simulating the failure of various software components, we can assess whether the remainder of the system can tolerate it.

I will illustrate this by using Advanced IBM Unix's (AIX) cos() function in this example. Note that AIX's cos() is a fine-grained COTS utility for which we do not have access to the source code:

```
double cos(double x)
```

This declaration indicates that the **cos()** function receives a double integer (contained in variable x) and returns a double integer. Because of C's language constraints, the only output from **cos()** is the returned value, and hence that is all that IPA fault injector can corrupt.

To see how this analysis works, consider an application that contains the following code:

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

The goal, then, is to determine how the application will behave if **cos()** returns incorrect information. To do so, the return value from the call is modified:

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

Note that IPA is more than just an

interesting research idea. It has been used successfully in a variety of critical software systems that required better techniques to perform impact analysis and assess potential interoperability conflicts; those case studies are compiled in [5].

Composing *ilities*

Clearly, today's COTS components are much more substantial in functionality and complexity than the previous AIX example. One of the real problems in composing today's COTS components that have advanced functionality and complexity is combining large components that each have varying quality-of-service (QoS) attributes into one component that now inherits a new set of QoS attributes.

Much of the work from the past 10 years into component-based software engineering (CBSE) and component-based development (CBD) has dealt with functional composability (FC). FC is concerned with whether the following is true:

F(A) _ F(B) = F(A _ B)

where,

_ is some mathematical operator

That is to determine whether a composite system, F(A _ B), is created that has the desired functionality after joining A and B. Therefore, A and B now have a way to communicate, whether one-directional or bidirectional. Instead of acting alone, they now act together as one unit.

But, increasingly, the software community is discovering that FC, even if it were a solved problem, is not mature enough for other serious concerns that arise in CBSE and CBD such as the problem of composing *ilities*. *Ilities* are non-functional, QoS properties of software components and they define characteristics such as security, reliability, fault-tolerance, performance, availability, safety, testability, survivability, maintainability, etc. The properties, as well as others, are what ultimately determine whether the software is well-behaved or not.

The problem stems from our inability to know a *priori*, for example, what the security of a system composed of two components – A and B – will be even if we have knowledge about the security capabilities built into A and knowledge about the security capabilities of B. Why? Because the security of the composite system is based on more than just the security of the individual components.

For example, suppose that A is an operating system and B is an intrusion detection system. Operating systems usually have some level of authentication security built into them, and intrusion detection systems have definitions for the types of event patterns that likely warn of an attack. Thus the security of the composition of these two components depends on the security models built into the individual components.

But even if A has a worthless security policy or flawed implementation, the composite can still be secure. How? By simply making the performance of A so poor that no one can gain access, i.e., if the intrusion detection system is so inefficient at performing an authentication, then in a strange way, security is actually offered. And if the implementation of A's security mechanism is so unreliable that it disallows access to all users, even legitimate ones, then strangely security is again increased. While these last two examples are not a reasonable way to achieve higher levels of system security, both do actually decrease the likelihood that a system will be successfully attacked.

Using the same example of A and B, this time assume that A provides excellent security and B provides excellent security. The usefulness of B's security mechanisms is a function of calendar time because new threats and ways to intrude are always being discovered. So even if you could create a scheme that determines the following:

Security(A) _ Security(B) = Security (A _ B)

the security offered by B is always a function of which version of B is being composed with A, what recent new threats have arisen, and how many of these new attack patterns have been built into B.

So the question then comes down to which ilities, if any, are easy to compose? The answer is that there are no ilities that are easy to compose, and some are much harder than others. Further, there are no widely accepted algorithms for how to do so.

For this problem applied to reliability, consider a 2-component system in which component A feeds information in B, and B produces the output of the composite (two components in series). Assume that both components are reliable. What can we assume about the composite's reliability? While this information certainly suggests that the composite system will be reliable, it must be recognized that components (which were tested in isolation for their individual reliabilities) can suddenly behave unreliably when connected to each other, particularly if the isolated test distributions did not at all reflect the type of information that A will be sending to B.

This brings us back to the importance of understanding the environment and operational profile for each component. Further, there can be non-functional behaviors that cannot be observed nor manifest themselves until after composition occurs. Such behaviors can undermine the reliability of the composition. Finally, if one of the components is simply the wrong component – although highly reliable – then naturally the resulting system will be useless.

In addition to reliability and security, one *ility* that, at least on the surface, appears to have the best possibility of successful composability is performance. But even that is problematic from a practical sense. The reason stems from the fact that even if a performance analysis was performed on a single component, the practical consequences on that component's performance after a composition with another component may depend heavily on the hardware and other physical resources. That could require that different hardware variables might need to be dragged along with a certificate that states only minimal, worst-case claims about the performance of the component.

For example, in the pharmaceutical industry, drugs come with endless warnings/rules as to who should take them and who should not. In the discussion here, the hardware variables are the various other medical circumstances of an individual slated for a particular drug (liver problems, heart problems, etc.). Clearly, this complex issue creates serious pragmatic difficulties and again takes us back to the need for a precise definition of what is the real environment of a software component [2, 6]. Today, the definition of what is the real environment of a component is an open question, needing new thinking and new research efforts [7].

Note that non-functional behaviors are particularly worrisome in COTS software products. Non-functional behaviors can include malicious code (Trojan horses, logic bombs, etc.) and any other behavior or side effect that is not documented.

Another worrisome problem facing CBSE and CBD is hidden interfaces. Hidden interfaces typically are channels through which application or component software is able to convince the operating system to execute undesirable tasks or processes.

Interestingly, IPA can partially address the issue of detecting hidden interfaces and non-functional behaviors by forcing software systems to reveal those behaviors after the input stream of a COTS component receives corrupted input. Injecting corrupt information into a component can possibly force it to execute different (and rarely executed) code paths, flushing out behaviors that would not normally occur with uncorrupted information.

The reason that IPA can only be partially successful in doing so is that IPA is a function of (1) the number of corrupted inputs used, and (2) the number of executions of the software. Just as *exhaustive testing* is the only way to guarantee correctness via testing, IPA is limited in the amount of analysis it can perform to detect hidden interfaces and non-functional behaviors. In short, you cannot test every potential combination of ways that you can inject corrupted data to see what results, and where those results propagate.◆

References

1. Voas, J., and C. Vossler. "Defective Software: An Overview of Legal Remedies and Technical Measures Available to Consumers." Academic Press 53 (2001): 451-497.
2. U.S. Patent No. 6,862,696: System and Method for Software Certification. 1 Mar. 2005.
3. Voas, J. "Would the Real Operational Profile Please Stand Up?" IEEE Software 17.2 (Mar. 2000): 87-89.
4. Friedman, M., and J. Voas. Software Assessment: Reliability, Safety, Testability. New York: John Wiley & Sons, 1995.
5. Voas, J., and G. McGraw. Software Fault Injection: Inoculating Programs Against Errors. New York: John Wiley & Sons, 1998.
6. Voas, J. "Certifying Off-the-Shelf Software Components." IEEE Computer 31.6 (June 1998): 53-59.
7. Whittaker, J., and J. Voas. "Towards a More Reliable Theory of Software Reliability." IEEE Computer 33.12 (Dec. 2000): 36-42.

About the Author

Jeffrey Voas is director of Systems Assurance Technologies at Science Applications International Corporation (SAIC). Before joining SAIC, he was the chief scientist at Cigital. Voas has been highly active in the software engineering research community for over 15 years. He has served on numerous journal and magazine editorial boards, written more than 125 papers, co-authored two books, and is currently the 2005 Institute of Electrical and Electronics Engineers Reliability Society president.

SAIC
Crystal Gateway #4
200 12th ST S STE 1500
Arlington, VA 22202
Phone: (703) 414-3842
Fax: (703) 414-8250
E-mail: jeffrey.m.voas@saic.com