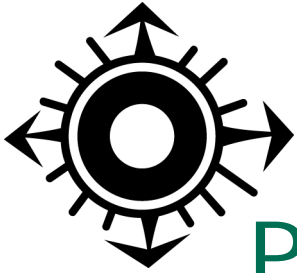Jeffrey Voas

Protecting against What?
The Achilles Heel of Information Assurance

Many have long regarded software assessment as a way to determine the correctness of software. Formal methods attempt to build in correct behavior. Techniques such as formal verification and testing attempt to demonstrate, either formally or empirically, that the software computes the specified function—whether or not the specified function is correct.

Note several subtleties here. First, to employ these techniques, we need a definition of correct behavior. Without an accurate definition of what we want, we cannot confidently label an information system as defective. Second, the predominant goal of software assurance has been to demonstrate correct behavior. But as we all know, correct software can still kill you. Correct and safe behaviors can conflict since safety is a system property while correctness is a software property. We must merge these two properties if we ever hope to realize information assurance.

Information assurance is similar to software assurance but covers a broader set of information integrity issues, such as information security, privacy, and confidentiality. For example, if a system can thwart attacks, whether malicious or simply unfortunate, and still provide accurate information on demand, then it provides some degree of information assurance. Information assurance also includes the traditional software "ilities" (as they are called), such as software safety, software security, reliability, fault tolerance, correctness, and so on. Put simply, information assurance is accurate enough information that is available on demand for a given application or situation.

CULTURE DETERMINES BEHAVIOR

The software "ilities" offer an interesting mix of complementary and conflicting behaviors. It is nearly impossible to build a complex software program that has a high degree of one "ility," let alone several of them. From a software perspective, the key to attaining software assurance is to

1. determine the important "ilities" before system design,
2. define precisely those software behaviors that would violate each "ility,"
3. design and implement the software to not exhibit behaviors defined in step 2, and
4. test the system to determine the effectiveness of step 3.

When the "ilities" are mutually exclusive (for example, security requirements thwart performance requirements), we must perform trade-off analysis to prioritize them. While this might seem more intuitive than revolutionary, clearly a key reason why the software industry is rumored to be 65-percent inefficient is that practitioners do not adhere to these basic steps—in particular, step 2.

The reason for this seems to be cultural—it is easier to define what we want a system to do than to define what we don't want it to do. When the information system controls processes or information of national security importance or that affects human life, then clearly we cannot allow certain events to occur. The developers must be told this.

I recently discussed with a system integrator for a large safety-critical system how he could be confident that the COTS software components he plans to use are of high enough quality to not cause the system to experience unsafe behavior. The system is a huge air-traffic control project that will impact millions of lives during its decades of operation. The integrator asked how I would go about qualifying the COTS software and suppliers.

I answered that I would start by defining what the complete system was "not supposed to do." That is, I'd start with the results of a hazard analysis for the complete system and then work my way down to the individual components. With this information in hand, I would combine static and dynamic impact analyses to see whether the COTS software is likely to violate those behaviors (defined in step 2). COTS software could very well output states that place the

system into those hazardous modes.

After further discussions with the integrator, I also discovered that not all components in question were customized for safety-critical applications—many were shrink-wrap data processing offerings. This begged a system-level analysis that would first define what the system was not supposed to do, then decompose that further into what the COTS software components could not be allowed to do.

My comments puzzled the integrator. I quickly learned that he had never thought of writing down the unsafe system behaviors. Yet he magically expected to be able to partition competing COTS products into two groups:

1. those that would cause an unsafe system problem, and

2. those that would not.

Unfortunately, this scenario plays out far too frequently in practice. Fortunately, if developers do perform step 2 as best they can, three techniques can then help them assess the level of information assurance provided. These assessment techniques can also be used to increase assurance.

1. First, system-level testing (using the operational profile (J.D. Musa, *Software Reliability Engineering*, Wiley, New York, 1998) as well as off-nominal profiles (J. Voas and K. Miller, "Predicting Software's Minimum-Time-to-Hazard and Mean-Time-to-Hazard for Rare Input Events," *Proc. Int'l Symp. Software Reliability Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1995) can determine whether the software exhibits behaviors defined in step 2.

2. Second, fault injection can be used to assess how well the software thwarts malicious and non-malicious anomalous behaviors. By thwart, I mean the software does not exhibit those behaviors defined in step 2.

3. Third, developers can use fault injection as a sort of "hazard mining" to potentially augment those behaviors defined in step 2 (J. Voas, "Hazard Mining," *2nd IEEE Workshop on Application-Specific Software Eng. and Technology*, Mar. 1999). Clearly, any initial and even subsequent hazard analysis will likely be incomplete. Hazard mining is a late-life-cycle technique that allows system designers to gain assurance that the list of undesirable system behaviors they have protected against is complete enough to offer adequate information assurance.

After several hours of discussions with the integrator (and personally knowing nothing about air-traffic control systems), we had enumerated several key events that are fully unacceptable and may enable a controller to make a wrong decision, among them

♦ radar screens going blank,
♦ radar screens losing planes, and
♦ radar screens showing incorrect heading

Welcome to the first Quality Time column of 1999. Steve McConnell has allowed me to pick up the reins from Shari Lawrence Pfleeger. I would like to congratulate Shari for the excellent columns she provided to *IEEE Software*'s readers. And I wish to thank Steve for giving me the opportunity to continue this forum.

As the new Quality Time editor, I seek to challenge practitioners to strive for higher levels of quality in their software, and to challenge researchers to ponder and solve "real" quality problems. With the age of "information assurance" upon us, quality must be our highest priority. Since achieving higher levels of quality carries greater costs, we will attempt to provide proof-positive for why and when to seek higher levels of quality, and how to take those arguments to management.

This column will also serve to inform the software community about new laws and pending legislation that directly affects software quality. I will work with Larry Graham, new editor of the SoftLaw column, to ensure that such pieces appear in the proper column should similarities arise. As you know, the software and IT landscape is rapidly changing, so look for updates on laws that may alter what you consider as "good enough" software.

Finally, I depend on your feedback. Please tell me your opinions and what topics you'd like to see addressed. I'd like for Quality Time to be one of the first things you look for when a new issue arrives. And that won't happen without lots of two-way communication.

—*Jeffrey Voas, Quality Time editor*

information (for example, showing the plane going in the wrong direction).

As we discussed the possibilities, we realized that there was not an enormous set of undesirable behaviors for the new ATC system but rather a core set of totally unacceptable fundamental failure modes. Other failure modes could be deemed acceptable since human judgment and intuition (as well as other backup systems) could overcome such failures.

SPELLING OUT QUALITY

I titled this column "Protecting against What?" because far too often practitioners set unattainable, "tongue-in-cheek" quality goals that do not spell out those events that must not occur. If practitioners do spell out such events, they can justifiably argue that minimal information assurance has been achieved once it can be demonstrated that the system cannot exhibit the behaviors defined in step 2. I say minimal because it is still possible that the information system can behave in unsafe ways that we failed to define (and therefore protect against). If practitioners opt not to address step 2, certificates boasting assurance cannot be justified. ❖

Jeffrey M. Voas is cofounder and chief scientist of Reliable Software Technologies. He holds a PhD in computer science from the College of William and Mary. Contact Voas at jmvoas@rstcorp.com.