

Defensive Approaches to Testing Systems that Contain COTS and Third-Party Functionality

Jeffrey Voas
Reliable Software Technologies
21515 Ridgetop Circle, Suite 250
Sterling, VA 20166
jmvoas@rstcorp.com
Tel: 703.404.9293
Fax: 703.404.9295

Abstract

Most systems today are composed of hardware components, COTS software, and custom software. When a system fails, a confusing and complex liability problem ensues for all parties that have contributed software and hardware functionality to the composite system. This paper presents a consumer-oriented methodology for predicting what impact on system quality a particular Commercial-Off-The-Shelf (COTS) software component will have.

When the result computed by the custom software causes a system failure, it becomes necessary to track down why that result occurred. If it is because of a logical defect in the custom software, then the vendor of the custom software is liable. If the result occurred because of a failure of a COTS software component (upon which the custom software was dependent for information), then the COTS vendor should be liable. Regardless of how these events might get argued in a court case and who would prevail, those persons responsible for integrating custom and COTS software together should take pro-active steps to ensure that all safeguards against COTS software failures have been taken. That is clearly their best legal defense strategy. This paper presents methods that provide those safeguards.

1 Introduction

The legal and professional costs of proving negligence by vendors of Commercial-Off-The-Shelf (COTS) could easily wipe out small system integrators that rely on COTS functionality. Further, proposed legislation in Uniform Commercial Code Article 2B (Draft) [3] and California Assembly Bill 1710, introduced January 28, 1998, if written into law, will make it even more difficult for system integrators to successfully sue vendors of defective software. To lessen the probability that

faulty COTS software will put system integrators into this situation, this paper recommends methods for assessing and mitigating the likelihood that COTS software components will negatively impact a system's dependability.

Fortunately, the methods we will recommend can be applied before any candidate COTS components are integrated into a system. Also, these methods do not require that a COTS software vendor disclose information concerning their internal development process. These methods provide an integrator with the luxury of taking pro-active steps against COTS components that might cause problems.

Because software liability is a hotly contested subject, it is timely to discuss ways to test for how much damage COTS software failures might cause and how to take protective measures before the software is released. This paper is an overview of earlier papers that have discussed ways to qualify COTS dependability and COTS liability [5, 8, 4, 7, 6].

2 Why COTS Software In the First Place?

The cost of software design and development could be significantly reduced if we had a widely-used software component industry. Even the best programmers only churn out 10 lines of code per day, which for systems such as cellular phones (that now have around 300,000 lines of code in them), have made custom software development very expensive. If, for example, 100,000 lines of code could be commercially purchased, 10,000 programmer-days of effort could be saved, allowing products to enter markets sooner.

Besides the time savings, another "win win" situation for both the consumers and producers is possible. From a consumer's standpoint, if a "world-class programmer"-day costs \$500, licensing 100,000 lines of code would result in a \$5,000,000 savings (minus the costs to license the functionality). From a software vendor's standpoint, if \$1M was an acceptable licensing fee for these 100,000 lines of code, then after 5 licenses, the vendor would break even, and on the 6th license would see profit.¹

Both government and commercial organizations are already gearing up for COTS software functionality. For example, government guidelines and standards have already been released by the US Federal Aviation Administration, US Department of Defense, National Research Council of Canada, US Food and Drug Agency, Electric Power Research Institute, and the US Nuclear Regulatory Commission.

The futuristic vision of a software component marketplace filled with competing COTS software components has good precedent. Traditional manufacturing is based on the concept of components. Software development is analogous to traditional manufacturing, particularly since software systems are composed of smaller software objects. What is lacking in the software industry that is ubiquitous in manufacturing is the ability to confidently swap components in and out of systems. By "confidently", we mean knowing that a replacement component is as dependable or

¹ Admittedly, this simple analysis has ignored many important market-based variables, but it does hint at the basic business principles that are driving consumers and producers toward a COTS marketplace.

better (in terms of logical quality) than the replaced component [9]. If this were possible, software component commerce would flourish, thus decreasing the costs of designing and repairing systems.

Key reasons for concern about components are: (1) a lack of knowledge concerning how good newly acquired components are, and (2) not knowing how well the system will tolerate the components. Even knowing that a component is highly dependable does not guarantee that the component will operate acceptably in a particular environment.

The COTS dilemma stems from the fact that COTS components are “black-boxes.” We say black-boxes because COTS components are usually delivered as *executable* objects (with licensing agreements that forbid decompilation back to source) as opposed to *source* objects. Without the source there is no way to know precisely what is going on inside the component. Certainly information going into a COTS component can be observed as can the information coming out. But what else is happening in the box is not known. And it is the unknown that engenders the fear.

Other than subjective factors such as “word-of-mouth” reputation and price, there are few other ways to know how dependable a component is. If a component is new, little historical information about it will be available. Further, it is possible to be lulled into thinking that more expensive components received better testing and thus are more dependable. The opposite could also be true: a less expensive component that has experienced more usage may actually have higher dependability.

2.1 COTS and National Security

COTS systems cause great dependability fears. Probably nowhere is the concern greater than to information system security. The US Government considers the reliance of our military and national information infrastructure on public systems (such as the Internet and the telephone system) as severely compromising to national security. Currently, the US Government is spending billions of dollars in search of solutions to this vulnerability [2].

Biological systems use genetic diversity to enhance their survival. Each individual of a species is slightly different from another individual. The diseases that one individual is susceptible to may not damage another. This diversity increases the probability that a species will not be completely wiped out when epidemics occur. In information systems, however, we see the reverse trend occurring. We see less and less diversity being available, particularly in operating systems, due to the mainstream cry for standards and interoperability. In operating systems, we are converging towards two main platforms: UNIX and Windows. Operating systems are probably the most important of all COTS components today. Further, we are converging toward a handful of Web browsers, and this number, too, is likely to get smaller in the coming years. Because of this lack of diversity, we are all susceptible to the same types of attacks and vulnerabilities. And because our operating systems are off-the-shelf, we may also be deficient in knowing everything going on in them and hence taking the appropriate action to protect ourselves.

The issue here is the *covert channel problem*. An executable component (other than the OS) may be making calls to the operating system that it is not supposed (and known) to. To determine whether this is happening requires a watchdog utility that has access to operating system level functionality. Tracking global environmental events requires the ability to keep track of the entire

system. For example, it will probably be useful to monitor `DeviceIoControl` function calls. Not only will such calls need to be tracked, but isolating exactly who (or what component) is doing the calling is also required. This approach amounts to trying to wrap the operating system in order to see every request that enters or leaves the operating system. The downside to this approach is that it is both expensive to develop the utility, and expensive to execute it when the operating system is deployed. Also, this scheme would need to be implemented for each unique operating system.

3 Assessing COTS Software Failure Impacts

The first step in our approach is to determine how a system reacts to incorrect information being passed to it from COTS software functions. After all, if a COTS failure does not negatively impact the system, then concern over the dependability of the COTS component may be unwarranted.

The technique that we will use here is termed *Interface Propagation Analysis* (IPA). It predicts the impact that a COTS component failure will have on a system [4]. IPA is a fault injection technique that simulates an interesting type of failure: the failure of a COTS component. This occurs as the system executes, allowing the system integrator to determine *a priori* whether COTS component failures will have severe system-wide consequences.

The process of performing IPA is quite simple. The interfaces that are responsible for sending information out of a component and into the remainder of the system are first isolated. Random data generators are placed at those interfaces. As information exits a component, the generators grab the information and corrupt (modify) it. That modified information is then handed over to the system in place of the original information. This provides an analysis of how badly the system behaves when artificially corrupted information is injected into the state of the system.

Recognize that the system may behave badly even when the unmodified information is carried across the interface and into the system. And that bad behavior may be regardless of whether a COTS component has actually failed. System-level testing, in theory, will determine whether this is likely to be true. But if component failures are rare, then it is unlikely that system-level testing will provide any insight. And if it is rare when a correct component output causes system-wide problems, then again system-level testing will provide little insight (unless, of course, enormous amounts of system-level testing are performed). So by forcing artificial component failures to occur, we can more quickly assess the tolerance of the system, even though we must always caveat our results with the realization that our injected failures were artificial.

4 Mitigating COTS Software Failures

After it is determined that a system cannot tolerate certain classes of artificial component failures (that were simulated by IPA), it is necessary to demonstrate that those artificial failures are not possible after the component is embedded into the software. If we mitigate the possibility of those failure classes occurring, then we can have more confidence in the component.

In [8], a mitigation methodology is provided. The methodology involves asking the component's vendor to perform: static fault tree analysis or backward static slicing (with testing). If the vendor does not do so, the component adopter can opt to wrap the candidate component themselves.

1. **Static Fault Tree Analysis (SFTA)** (Watson, 1961) can prove (using proof-by-contradiction) that certain failure classes are not possible from the component, including component failures caused after the component receives bad input data.
2. By **backward static slicing** [10] from those variables that were corrupted, you can isolate the slices that need concentrated testing. This provides an *argument* that the artificial failure modes cannot be created by the component as it executes in its natural environment. This analysis, however, does not formally prove this.
3. **Component wrapping** can filter the component's inputs, outputs, or both. By building a classifier heuristic from the artificial failure modes, we will not only be able to prevent the artificial failure modes, but we should be able to prevent real component failure modes with similar characteristics.

The problem with the first two of these is that: (1) it will be very difficult to convince the component's developer to spend the resources to prove that their component cannot cause failures that were generated randomly using a fault injection tool. Further, the set of artificial failure modes needing mitigation may be very large. This will further reduce the likelihood of cooperation by the component's vendor.

Therefore wrapping components using various classification heuristics that describe the problematic artificial component failure modes is probably the best approach to protecting systems from component failures. And this can be done without any help from the component vendor.

How this works is fairly straightforward. First, the potential adopter of a COTS component applies IPA to gather the set of intolerable artificial failure modes. These modes are then classified into a heuristic. This heuristic is then embodied into a wrapper that surrounds the component, checking to ensure that the component does not output states that fit the classification heuristic. Note that this heuristic-based wrapper is not foolproof. It can fail to detect undesirable states or misclassify acceptable states as undesirable. But for systems where failure is not an option, wrapping COTS components is currently the best that can be done to ensure that those components do not cause system-wide problems.

5 Liability Concerns from Bad Software

By now you might be wondering why you need to expend so much effort to ensure that someone else's software works. You might be thinking "if the COTS software fails, I'll sue." And that option may be available. But be forewarned. Early signs in the industry are suggesting that future laws will protect software vendors so effectively that suing vendors may be futile.

The November 1997 draft of Article 2B of the *Uniform Common Code* [3] provides serious liability protection for software vendors. In the United States, the Uniform Common Code is the series of laws governing all financial transactions that are not covered by written contracts. For example, you assume that when you buy a banana in a grocery store there is a banana under the skin. If there is not, the Uniform Common Code provides you with the right to demand your money back even though you never signed a contract when you purchased the banana.

Article 2B is a planned addition to the Uniform Common Code that handles “transactions in information.” It handles the laws for transactions relating to the “copyright industries.” While it sounds reasonable to finally have laws governing what is a fair legal transaction in software, Cem Kaner has pointed out that Article 2B has little protection in it for the consumer and instead is full of vendor protection. He writes that the consumer will ultimately suffer from bad software and not the vendors [1]:

“Article 2B denies the mass-market customer most remedies, even a refund of the fee charged by the publisher for calling to report problems. Customers are entitled to refunds only for “material” breaches of contract, and 2B redefines “material” to be narrower and less inclusive than the Restatement of Contracts. Article 2B denies the remedies even when the customer’s damages are caused by a defect that was known to the publisher before the product was released for sale, that the publisher chose not to fix and chose not to warn the customer about.

Article 2B lets publishers pick what law will govern their sales and where customers can sue them. There are no geographical restrictions and no requirement of any relationship between the law, the forum and any party or aspect of the transaction. Small claims court actions will be unavailable (when exclusive jurisdiction is given to a higher-level court) or prohibitively expensive (in a state or country far, far away). Article 2B is a forum shopping gone wild, and available only to the publisher. There are slight restrictions on this if the customer is a “consumer” who uses the software for strictly non-business, non-professional purposes (a teacher who does research or writes assignments at home is not acting as a consumer, nor is the unemployed secretary who tries out some home-based network marketing scheme and uses a computer to manage her mailing lists and print fliers.) Few customers with meritorious cases will have the ability to bring a lawsuit against a publisher.

Article 2B settles the Battle of the Forms (the traditional problem of conflicting terms and expectations set between the publisher and the customer) by creating a new set of procedures that will ensure that the publishers wins the battle. The publisher gets the last shot, in the “mass-market license agreement,” which the publisher need not even make available to the customer until after the customer has paid for the product, taken it away, and started installing it on his computer. With extremely few exceptions, all of the terms in this “license” “agreement” will be fully enforceable against the customer as if he had reviewed, discussed, and signed a paper contract before the sale.”

Article 2B is not yet law in any state and this draft has not yet been finalized by its authors. If nothing else, Article 2B should force software consumers to realize that their legal protection

appears to be weakening. Consumers will hopefully demand tougher software measures such as the mitigation strategies recommended above. That is, consumers could employ these mitigation techniques as a way to counterbalance the lack of consumer protection afforded in the current draft of Article 2B (if it does become law).

In addition to Article 2B, we see other vendor-oriented (and thus consumer-unfriendly) legislation on the horizon. California Assembly Bill 1710, introduced January 28, 1998, if passed, will amend the California Civil Code to limit recovery for damages resulting from the Y2000 date transition. If this bill becomes law, in a lawsuit for a Y2000 defect, the plaintiff could recover only the costs of (1) damages that resulted from bodily injury, and (2) fixing the bug. The bill states:

“(a) Notwithstanding any other provision of law, in any action to recover damages resulting directly or indirectly from a computer date failure, including any action based on an alleged failure properly to detect, disclose, prevent, report on, or remediate a computer date failure, the damages that may be recovered shall be limited to either or both of the following, according to proof:

(1) Any damages resulting from bodily injury, excluding emotional injury, to the plaintiff proximately caused by the defendant’s conduct.

(2) Any costs reasonably incurred to reprogram or replace and internally test the relevant computer system, computer program or software, or internal hardware timer, to the extent those costs are incurred as a proximate and direct result of the defendant’s conduct.”

Here again, we see the purchaser of defective software ultimately having little legal recourse.

6 Summary

This paper has recommended methods for assessing whether a system can tolerate failures originating from external subsystems. When the answer has been “no”, we have provided recommendations for how to proceed with mitigation.

This paper has focused on COTS software subsystems. Because COTS software is often failure-prone, “defensive system designing” is prudent. This paper has also mentioned pending laws that appear to provide vendors of unreliable software the legal “loop holes.” That legislation favors software vendors at the expense of software consumers.

The vendor of a system is responsible for the dependability of the system, regardless of where they acquired different components. By keeping abreast of proposed laws and methods that thwart component failures, you increase your chances of success in the marketplace for your software products.

References

- [1] C. K_{ANER}. Article 2B is Fundamentally Unfair to Mass-Market Software Customers, October 1997. Submitted to the American Law Institute for its Article 2B review.

- [2] GENERAL J. J. SHEEHAN. A commander-in-chief's view of rear-area, home-front vulnerabilities and support options. In *Proceedings of the Fifth InfoWarCon*, September 1996. Presentation, September 5.
- [3] THE AMERICAN LAW INSTITUTE AND NATIONAL CONFERENCE OF COMMISSIONERS ON UNIFORM LAWS. Uniform Commercial Code Article 2B (DRAFT), November 1997.
- [4] J. VOAS. Error Propagation Analysis for COTS Systems. *IEE Computing and Control Engineering Journal*, 8(6):269–272, December 1997.
- [5] J. VOAS. An Approach to Certifying Off-The-Shelf Software Components. *IEEE Computer*, To appear in June 1998.
- [6] J. VOAS. COTS: The Economical Choice? *IEEE Software*, March 1998.
- [7] J. VOAS. Independent Software Measurement's Role In The Liability Puzzle. In *Proc. of FESMA'98*, Belgium, May 1998.
- [8] J. VOAS. Mitigating the Potential for Damage Caused by COTS and Third-Party Software Failures. In *Proc. of AQUIS'98*, Venice, March 1998.
- [9] J. VOAS. Software Component Dependability Assessment. *ACM Computing Surveys*, September 1998.
- [10] M. WEISER. Programmers use slices when debugging. *CACM*, 25(7):446–452, July 1982.