COTS Software Failures: Can Anything be Done?

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

Abstract

Software development is quickly becoming more of a process of acquiring software components and composing them than building systems from scratch. From a time-to-market perspective, this is ideal, but from a quality perspective, this is worrisome. This paper addresses steps that component integrators should follow before relying on someone else's software libraries and components.

Keywords

COTS, fault tolerance, testing, wrappers, fault injection

1 Introduction

The adage, "if you want something done right, do it yourself" is less of an option for software developers today than it was 10 years ago. Today's software systems are complex "systems of systems", and developers must accept the fact that substantial portions of these composite systems will be provided by other developers. Losing control over *every* aspect of a system's functionality may worry the parties that are legally liable for the quality of the complete system. Those parties need assurance that the components can tolerate each other.[1]

Software reuse has the potential to massively increase the rate at which information systems are built while reducing the costs of building these systems. Software reuse occurs in different ways, including: (1) purchasing Commercial-Off-The-Shelf (COTS) "generic" software, and (2) reusing one's own software modules from project to project through shared libraries.[2] But each of these methods run the risk that the complete system will suffer from problems caused by the reused or acquired software. This paper presents a methodology for predicting whether this is likely to occur as well as approaches for reducing this likelihood.

[1] When two distinct software components interact correctly, this is referred to as *non-interference* between the components.

[2] There is a special case to (1) and that is procuring custom software functionality from a third-party vendor. Here, the software may or may not be new, but it presents the same risks of reused software from a quality standpoint, and thus the methodology that we will propose here will apply to this situation.

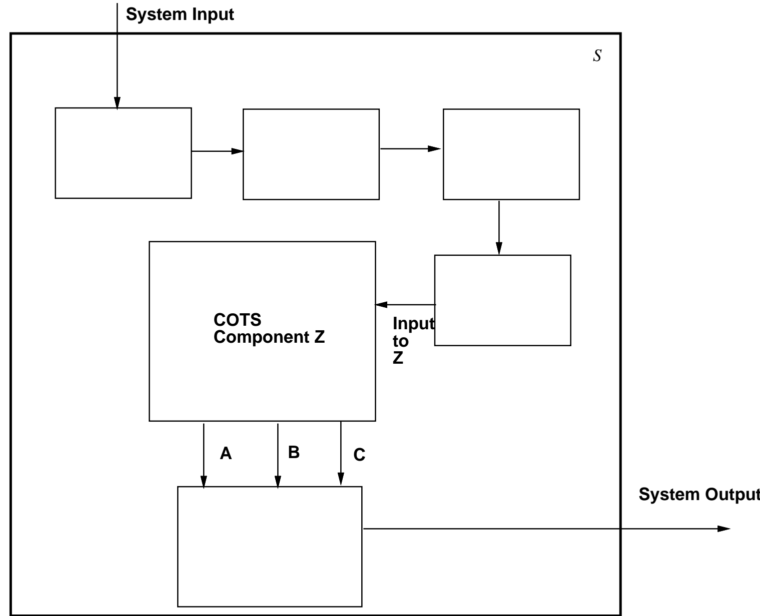

Figure 1: Typical System S that includes a COTS Component, Z.

Software reuse is easy to justify. The cost of software design and development could be significantly reduced with more reuse. 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 [1].

Reductions in development costs and decreases in time-to-market encourages reuse. But reuse is not a panacea. The Ariane 5 disaster demonstrated that fact [2].

Reuse enables bigger and more complex software systems. According to Reme Bourguignon, V.P. of Philips in Holland (as well as numerous public quotes made by Airbus officials), the amount of software embedded in a typical device is doubling in the number of lines about every 18 months, and much of that software is reused. Consider that today, even the best systems still have software defect densities of around 3-6 faults per KSLOC (if you count every fault, no matter how small it may be) [3]. Interestingly, that rate has held constant for the last two decades, regardless of the shift to object-oriented technology, software reuse, automated debuggers, better test tools and compilers, stronger type safety, etc. The combination of fixed defect densities and larger systems suggests that the reported crisis (See W. Gibb's article, "Software's Chronic Crisis", *Scientific American*, 1994) is very real.

This implies that if our industry cannot overcome this naturally occurring phenomenon of fault densities of 3-6 faults per KSLOC, then our industry must get better at diminishing the effects of the faults. The methodology proposed here, which mitigates the potential of dangerous program states occurring during the execution of a system that employs COTS functionality is one step in that direction. As it turns out, this methodology is particularly amenable to generic software components that are expected to operate in a wide variety of environments or throughout a family of products.

2 Methodology

Our methodology is based on the premise that defect-free logical systems is an oxymoron. Further, we dismiss the idea that component developers will ever guarantee absolute correctness of components. Even if they were to guarantee such via proofs-of-correctness, there is no guarantee that a component will not interfere with the remainder of the system, causing the system to have lower overall quality.[3] Thus we assume that each software component will have failure modes. We assume that these components will be fairly course-grained and much larger than traditional objects. The goal is to determine the severity of the component failure modes on the system and thwart those that are intolerable.

To illustrate our approach, consider Figure 1 which shows an information system, S, that contains a COTS component Z. Z has three different output variables: **A**, **B**, and **C**. To determine which failure modes from Z cannot be tolerated by S, fault injection [6, 5] forcefully corrupts (modifies) the information stored in **A**, **B**, **C**, or some combination of these. (Throughout the remainder of this paper, we will assume that only **A** is corrupted during fault injection to simplify the discussion. As shown in Figure 1, we denote this modified information as **A'**.)

Note that **A'** is not produced naturally by Z. **A'** is artificially corrupted information that is exiting Z and that is introduced into S using fault injection processes.

Naturally occurring corrupt information exiting Z may or may not cause S to produce undesirable output. Fault injection analysis allows us to better understand which of these two situations is more likely by studying how artificially corrupt information propagates. This provides a nice basis from which to reason about how actual corrupt information will impact S.

Undesirable output information from S is detected in our method by an assertion that monitors S's output. This assertion contains definitions for what are acceptable outputs for S as opposed to unacceptable outputs. These assertions should be traceable back to the specification or requirements for S.

By corrupting the information stored in **A**, we discover corrupt output states that cause S's assertion to fire.[4] Assertions fire when S produces undesirable outputs (where "undesirable" is defined with respect to the input that S received). Undesirable can be unsafe, vulnerable, incorrect, etc. The goal then is to know that Z cannot produce these corrupt states.

By artificially corrupting the results returned by a COTS component, we find outputs that we do not want the COTS function to produce. The key question then becomes: *are there inputs to the component that will cause the component to naturally produce those outputs that fault injection has determined are intolerable to the system?* If the answer is "no", then we have confidence that the component is well-suited for the system. If the answer is "yes", there are steps that need to be taken before Z is incorporated into S. This paper suggests several well-known techniques that may be able to answer this question.

3 Mitigation Strategies for Z

From this point forward, we will assume that fault injection was able to uncover at least one corrupted state that fired S's assertion. We will

[3] Composing two correct components together does not guarantee a correct composite. This is due to the possibility of *interference*.

[4] It is unlikely that we will discover all of these during fault injection analysis, but even if only a few are found, that information can be used in order to add robustness to the system.

now suggest mitigation strategies that address the above question.

At this point, it is premature to abandon Z, but we should be cautious. We do not have proof that Z will behave in a manner that is identical to those behaviors that were employed during fault injection (that resulted in intolerable sytem outcomes), but likewise, we have no guarantee that the component will not.

To determine whether a component can naturally produce those **A'**s, we propose a set of mitigation strategies. In certain cases, these mitigation strategies will fail to resolve our dilemma, as there are undecidable problems that cannot be overcome. These strategies place the onus on the component vendor to demonstrate the integrity of his or her components. After all, it is the component vendors that will profit if the components are adopted. If the component vendor fails to provide this assurance, there is still one other mitigation strategy that can be performed by the party that wishes to adopt the component. This strategy does not require effort from the component's developer.

Our approach involves performing fault injection first, and then performing the mitigation processes. For the vendor of a suspicious component, static fault tree analysis, backward static slicing (with testing), and wrapping are three mitigation strategies that will enable a vendor to show the integrity of their components:

1. Static Fault Tree Analysis (SFTA) can prove (by contradiction) that certain failure classes are not possible from Z, even if Z were to receive bad input data from the remainder of the system.

2. Backward static slicing from those variables that were corrupted by fault injection (and caused S's assertion to fire) can establish those slices that need concentrated testing. This provides an *argument* that those failure modes in **A'** are not possible from Z, but this is not a *proof*.

3. Wrap Z to either filter Z's inputs, outputs, or both. (Since all we have at this point are **A'**s, wrapping outputs makes more sense.)

Component vendors need to perform only one of these process to provide the necessary assurance. The reason that component developers are the best candidates for performing the analysis is that these analyses require access to the internals of the component. The adopter for the component will likely not have that information. If the developer is unwilling, wrapping can be performed by the system integrator, without help from the component vendor and without access to the code in Z.

We will now go through each of these mitigation strategies in more detail and how they address our key question. In a sense, our approach allows an adopter to query a vendor for particular guarantees that the adopter needs for their specific system.

3.1 Static Fault-Tree Analysis on the Component

Static fault-tree analysis assesses the causal relationship between events in a process. Software fault-tree analysis is simply the application of static fault-tree analysis to software. It was brought over to software engineering from systems safety engineering.

Those **A'**s that caused S's assertion to fire will be the top events in the fault-tree. These events will be OR-ed together. The underlying events represent: (1) failures of subcomponents of Z, or (2) failures of external components upon which Z depends for input information.

Fault-trees allow a developer to prove that type (1) and (2) failures cannot produce those

corrupt outputs from Z that caused S's assertion to fire. If during this "proof" process a branch exists that does not result in a logical contradiction, then the analysis has uncovered events that can lead to undesirable outputs from Z. This means that there are harmful outputs from Z that must be protected against. (Section 3.3 presents an approach for doing this.)

3.2 Slicing and Testing the Component

Besides fault-tree analysis, the component vendor has the option to perform slicing and then perform slice testing. Slicing and fault-tree analysis are both techniques that perform impact analysis, but the processes that they employ to study casual relationships are quite different. Slicing and slice testing, unlike fault-tree analysis, does not prove that the output events are impossible. The results from slicing and slice testing are statistical. Even though these approaches to impact analysis are different, the results from these techniques demonstrate that certain output events are unlikely. And this is precisely what we need to mitigate the potential of those outputs that fault injection has warned of.

After the static slices are discovered, the slices must be thoroughly tested to determine whether those **A'**s were even output from Z.[5] (This is referred to as *slice testing*.) Note that generating test cases to exercise a slice can be a difficult task, and like almost all problems that involve test case generation for code coverage, it can suffer from undecidability problems.

In summary, if those **A'**s do not occur during slice testing, confidence in Z for S is increased. If those **A'**s do occur, wrapping of Z should be considered. (See Section 3.3).

[5] It is out of the scope of this paper to define "thorough", but as a minimal condition, all of the code in the slice should be exercised.

3.3 Wrapping the Component

If the component vendor does not provide convincing evidence that Z cannot produce those **A'**s using the previous two mitigation strategies, there is still another alternative that the component adopter can undertake: software wrappers. Software wrappers are a family of technologies that force components to behave in specified, desirable ways. A software *wrapper* is a software encasing that sits around the component and limits what the component can do with respect to its environment. (There is also another type of wrapper that limits what the environment can do to the component.)

To accomplish this, two different wrapper approaches can be used. One approach ignores certain inputs to the component and thus does not allow the component to execute on those inputs. That clearly limits the component's output. The other approach captures the output before the component releases it, checking to ensure that certain constraints are satisfied, and then only allowing those outputs to result that satisfy the constraints.

Recall that fault injection told which outputs from Z were undesirable (with respect to the system state of S for those corrupted outputs). Wrappers should be based on that information. One option for accomplishing this is to build tables that contain undesirable corrupted states (along with the system states for which they are undesirable). If these states are seen in the future, the wrapper can halt their being output. Another option is to use artificial intelligence approaches to build heuristics that can classify the results of fault injection, and if outputs that satisfy the heuristics are observed, they too can be thwarted.

4 Conclusions

It as been predicted that software component catalogues are "the future." Catalogues will enable consumers to simply read through a listing of components, select those of interest, and plug them in. Clearly this is futuristic thinking, but how far away we are from this is unclear.

Component catalogues are instrumental in designing hardware systems. In electrical engineering, designers first discover what components are available, and then they design the remaining system around those parts. In our opinion, if component-based software is to evolve into a catalogue industry, it will begin by component consumers demanding specific warranties about component behaviors from component suppliers. As components mature, these guarantees will become more generic, allowing for a more "off-the-shelf" (as is) mentality. From here, software component catalogues can be built.

In the current state of our industry, it is still very difficult to produce highly reliable *custom* software. Consider that this is true after 30+ years of experience in building software. Given this, it is implausible to believe that we are yet in a position to expect that *generic* software components will perform better in an unknown environment than custom components that are specifically built for a known environment.

Admittedly, few component producers will cheerfully perform our recommended mitigation analyses. It may fall back onto the shoulder's of the adopter to perform tasks such as wrapping if the component vendors refuse. However, if system integrators were to universally demand such actions, component developers will have less say in the matter. And given the pending "consumer-unfriendly" software legislation such as Uniform Commercial Code Article 2B [4] and California Assembly Bill 1710, introduced January 28, 1998, adopters may be forced to.

References

[1] S. Baker, G. McWilliams, and M. Kripalani. "Forget the Huddled Masses: Send Nerds", Business Week, July 11, 1997.

[2] Prof. J. L. Lions. Ariane 5 flight 501 failure: Report of the inquiry board. Paris, July 19, 1996, available at `http://www.cnes.fr/actualites/news/rapport_501.html`.

[3] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability Measurement Prediction Application*. McGraw-Hill, 1987. ISBN 0-07-044093-X.

[4] The American Law Institute And National Conference Of Commissioners On Uniform Laws. Uniform Commercial Code Article 2B (DRAFT), November 1997.

[5] J. Voas. Error Propagation Analysis for COTS Systems. *IEE Computing and Control Engineering Journal*, 8(6):269–272, December 1997.

[6] J. Voas, G. McGraw, L. Kassab and L. Voas. A Crystal Ball for Software Liability. *IEEE Computer*, 30(6):29–36, June 1997.