An Application-Specific Approach for Assessing the Impact of COTS Components

Jeffrey Voas
Reliable Software Technologies

1 Introduction

The world software and services market is currently a 300 billion US dollar industry. The cost of software development can be significantly reduced by fostering a software component industry. In past years, most software was marketed as complete applications; today, we are moving toward an economy where it is hoped that components can perform dual purposes and thus be sold as stand-alone entities. When components can be marketed in this manner, they are referred to as "Commercial-Off-the-Shelf" (COTS).

Studies have indicated that software reuse is capable of a two to seven times increase in software productivity. Component-based software engineering that uses COTS components, however, is more than software reuse. This is because COTS-based software engineering saves the time and cost to build a component by immediately purchasing the component. The time-to-market decrease that occurs as a result cannot be overstated.

This design paradigm has precedent. Traditional manufacturing is based on the concept of components. Automobiles, radios, bridges, planes, etc., are made from hundreds or thousands of components. (Can you imagine the difficulty in trying to build a bridge as a single part?) 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 good or better (in terms of logical quality) than the replacement component. If this were feasible, software component commerce would evolve, which would decrease the costs of designing and repairing systems.

Today's software systems are "systems of systems." The semantic dependencies between software components are so complex that they can no longer be fully understood by a single person. Thus the engineering design paradigm known as divide-and-conquer must be employed. Because of this, it is likely that logical incompatibilities (component-to-component or component-to-environment) will exist after the system is in service. This paper describes a black-box analysis technique that studies how information flows (propagates) between software components. The technique ignores the source code that comprises the components. The technique forcefully corrupts the information that flows between components and observes what impacts the corruption had. Corrupting information in this manner simulates component failures or failures of the environment external to the component. By watching how corrupt information propagates, we isolate those components that cannot tolerate the failure of other components. This provides a dynamic determination of the system's sensitivity to individual components. By being independent of source code, the approach is particularly applicable to COTS-based systems.

2 Background

The key to unlocking the mysteries of software lies in understanding the states that are created during execution and how those states propagate through the system. Static metrics already exist that analyze control flow, data flow, coupling, OO designs, etc. These metrics have been formalized to allow developers need to understand "what can impact what." For example, by knowing how information might flow during execution, we can make intelligent decisions concerning which types of structural testing are more likely to find errors. Likewise during maintenance, we can use static information to determine if code changes in function X have any possibility to affect the output or other functions of the code.

But ideally we'd rather know "what *will* impact what" as opposed to "what *can* impact what." Dynamic measures of information flow can provide this more precise information. For instance, if function A calls B, a static measure would say that B can affect A. But this might not occur in practice because of the specific test cases used. If we study the dynamic dependencies between A and B with test cases, we can more precisely determine the semantic relationships between components. This paper provides a technique for doing so.

Studies in dynamic information flow can occur at various levels of granularity: between statements, modules, libraries, objects, etc. Because today's systems are so large, it is more practical to study information flow at higher levels of granularity. At the highest level, we view components as black boxes and study information flow across software components (across is defined as information flow from the input interface to output interface). This provides a determination of how COTS components will affect a system if the components get into trouble.

3 Commercial-off-the-shelf

The single greatest problem with COTS components is the lack of knowledge concerning their quality. After all, for COTS software to be profitable to its builders, it must be generic and have wide market appeal. COTS software is notorious for having much functionality. The problem is that these generic programs try to do too much, and do many of their tasks with marginal quality, as opposed to doing only a few tasks with high quality.

If seeing is believing, then what is COTS? Disbelieving? For COTS software, we do not know how the components were tested, what their failure modes are, how severe the failures are, etc. COTS is championed as a way to quickly and inexpensively build systems, but who knows if the improved time-to-market afforded by COTS software outweighs the potential quality risks?

Assuming that we still wish to rely on COTS software but need additional assurance in its quality beyond what manufacturers claim, we have two basic options: (1) perform black-box (reliability) testing of the COTS functionality in our operational environment, or (2) repeatedly simulate the failure of the COTS software inside of our system to see what impact those simulated failures have. Both of these approaches are proactive and provide a good defensive strategy, but the latter is probably more useful. By determining that a system is robust to problems originating from a COTS component, we can be less concerned about the failure of the COTS component. Further, to do (1), we need an oracle, but to do (2) only requires the ability to simulate component failures. If we do (1) and find that the components are reliable, that still does not tell how badly the system might behave if a COTS function misbehaves. Thus performing (2) is a more complete defensive approach since it is almost certainly true that the COTS component will fail. And performing both (1) and (2) would be the best approach if such can be afforded.

The remainder of this paper will focus on a method for performing (2). The approach we will use for

(2) is termed *Interface Propagation Analysis* (IPA). IPA is a fault-injection based technique for injecting "garbage" into the interfaces between components and then observing how that garbage propagates through the system. For example, if component A produces information that is input to component B, then we corrupt that information using fault injection techniques. This effectively simulates the failure of component A. After this corrupt information is passed into B, IPA analyzes the behavior of B (or components executed after B) to the information. IPA analyzes the behavior of a component by looking for specific outputs that the users wants the technique to be on the lookout for.

4 IPA

Interfaces are the mechanisms by which components communicate. Component-based systems can suffer from a variety of problems with the selected components:

1. the wrong components could be selected,
2. the components selected are the right ones but some or all of them are logically flawed resulting in incompatability problems,
3. the components are incorrectly composed, e.g., $A(B)$ instead of $B(A)$,
4. the complete software system or components of the system are fed corrupt input data during execution, or
5. some combination of these.

If (1) or (3), the interfaces are almost certainly carrying wrong information. Such a system will be so unreliable that the system will almost certainly never be deployed, and hence although this is disastrous from a quality perspective, we think the problem will be so obvious, it is unlikely to result in harm, because such systems are unlikely to be deployed.

But for (2) and (4) this is not necessarily true. These classes of problems can be far more subtle and may escape detection during testing and quality assurance. Fortunately, problems (2) and (4) can be simulated via IPA before such a system is deployed. To simulate (2) for some component X, the information returned by X will be intercepted before the information is handed over to the system. Fault injection mechanisms will first mutate the information in ways defined in [1]. After mutation, the modified information is passed onto the system and the system continues execution. The impact of this on the system's output is then observed. Simulating problem (4) is a similar process: information coming into a component is intercepted, mutated, and then execution is allowed to continue using the modified data. Again, the impact of this on the conponent or on the system can then be observed.

We view the information that an interface passes as a bit collection. We model failures as various corruptions on a bit-by-bit basis. Depending on the application, we can "flip bits" randomly or in pattern typical of failures in that type of application. The optimal degree of corruption (what percentage of bits will be negated) and the location of the corruptions are open areas of research.

To observe the impacts of the fault injection mechanisms, IPA must be told by the user what classes of internal states or outputs are undesirable. IPA then checks to see if any matches occur. Undesirable events can be defined by the user for system outputs or for the data flowing between components. By corrupting data exiting one interface, IPA can observe whether a succesor component produces corrupt information (because it was fed corrupt data).

Thus IPA provides an analysis of how errors propagate through a component-based system. This is a more expensive scheme but a more accurate way to study information flow than static schemes. Static

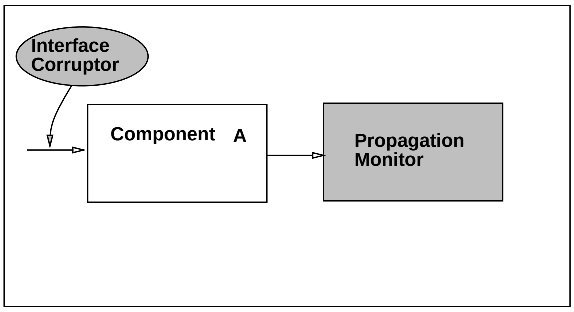

Figure 1: Studying "Propagation Across" Component A.

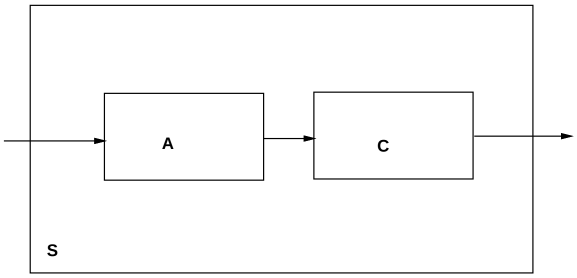

Figure 2: The system S made from components A and C.

schemes ignore input-dependent information, making them predictors of "what can impact what" as opposed to "what will impact what."

Observing the impact of fault injection is much less difficult than determining what to inject. Our approach for simulating component failures is to corrupt the information exiting a component. We can modify the data being passed or even delete the data being passed altogether (simulating the situation where the component fails to produce output). We also modify *global* information that the component has "write access" to. Global data that can be modified by a component might be accidentally overlooked as information that should be corruptible by IPA. But this is incorrect. More output can be incorrect because of a component failing than only the output transmitted via **return()** functions from the component. Thus global data must be a candidate for corruption during fault injection as well as considered as a candidate for observation when the effects of fault injection are observed.

Figure 1 shows an example of how IPA works. In Figure 1, we see the corruption of data traveling via the incoming interface of component A, and we place a watchdog monitor on the exit interface of A to detect if the output of A was affected. This studies how garbage propagates across A. In practice, this process requires executing A without the corruption, saving the output, re-executing A with corrupt information, and then comparing the two different output sets (which may include global states).[1]

4.1 Software Experiments In Combination

To get a feeling for the benefit of IPA, consider the system S composed of two components, A and C (shown in Figure 2). Now consider a new version of S in which a COTS component B is placed

[1]An alternative form of IPA exists that does not require executing A without the corruption, and instead looks for various types of output behaviors from the component that the user of the technique defines before analysis begins.

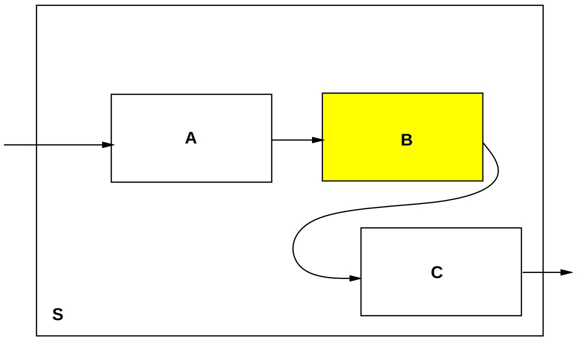

Figure 3: The system S augmented with a new reused component, B.

between A and C (see Figure 3). The developer considering this addition to S should already have information about the quality of B as a stand-a-lone component. But what does that mean in terms of the environment that A will force B to experience? Unless B has already been tested in this setting we cannot know. And if B is fairly generic this will likely not be the case. Thus for S, we need to run experiments to ascertain how S will behave after the addition of B.

Let f_y denote the fault injection mechanism for the exit interface of component y. For this example, IPA would examine the following different quality-related questions concerning B's addition to S:

1. $B(f_A(A))$: How will B react to bad input from A's output?
2. $C(f_B(B))$: How will C react to faults from B?
3. $C(B(f_A(A)))$: How will S react to faults from A?
4. $C(f_B(B(f_A(A))))$: How will S react to an interaction of faults between A and B?

Each of these experiments yields evidence about the effects that a defective A or defective B will have on S after B is added.

5 Conclusions

COTS components are more likely to combine successfully in a wide variety of software systems when the custom and COTS components can tolerate incompatibilities and errors in each other. For a given system, we can use fault injection techniques to examine the interfaces between components and determine whether the components are resilient to simulated faults. If the components resist simulated faults, they are more likely to resist (the inevitable) real faults, however this can never be guaranteed.

Acknowledgements

This work has been partially supported by DARPA Contract F30602-95-C-0282 and NIST Advanced Technology Program Cooperative Agreement No. 70NANB5H1160.

References

[1] J. Voas, G. McGraw, A. Ghosh, and K. Miller. Glueing together Software Components: How good is your glue? In *Proc. of Pacific Northwest Software Quality Conference*, pages 338–349, Portland, October 1996.