

Tolerant Software Interfaces: Can COTS-based Systems be Trusted Without Them?

J. Voas, F. Charron

Reliable Software Technologies Corporation
Sterling, VA 20166 USA

K. Miller

Department of Computer Science, Sangamon State University
Springfield, IL USA

Abstract

We have investigated an assessment technique for studying the failure-tolerance of large-scale *component-based* information systems. Our technique assesses the tolerance of the interfaces between component objects in order to predict how the software will behave if anomalous failures exit certain components and enter others. (Note that we are not talking about graphical user interfaces, but rather the mechanisms that link software components together.) These failures can originate from incorrect code, bad input data from a failed hardware devices, or bad input data from human operators. Our approach is applicable to systems for which source code is available, as well as systems for which no source code is known (e.g., systems composed from executable Commercial Off-The-Shelf (COTS) components), and addresses several of the larger problems associated with software maintenance.

1 Introduction

Large-scale information systems are made up of networks of closely related subsystems that may include distributed hardware and distributed software components. Even if a system is composed of correct components, the *interactive* complexity and coupling between components can cause unexpected errors to emerge from component interfaces. In our view, a system is “failure-tolerant” if it is capable of continuing its mission regardless of any *potential* anomalies that might possibly occur against it. Our solution for assessing the failure tolerance of large-scale systems is to assess the tolerance of the interfaces between components in order to predict how the software system will behave when actual anomalies arise, meaning when corrupt information enters into the *state* of an executing program and is passed between components. (Note that we are not talking about graphical user interfaces, but rather the mechanisms that link software components together.)

Suppose that you knew that the components of a proposed system were known to be failure-tolerant. The natural question that arises when combining these components into a larger system is: *how can we use our confidence in the failure tolerance of individual components to come up with a measure of*

failure tolerance for the entire system? Three key concerns regarding the quality of the composite involve: (1) the failure tolerance of the interfaces between components, (2) determining whether the failure of an individual component is likely to cause the failure of another component or, worse, the failure of the entire composite, and (3) the manner by which the components are composed (for example, does **A** call **B** or does **B** call **A**, and what difference does it make). The technique that we will report on here is able to analyze concerns (1) and (2). Analyzing (3) is out of the scope of this paper, however the approach that this effort describes might be extendible to this problem.

2 Approach

Information systems are the most complex artifacts in human history. Today's information systems include distributed hardware and distributed software components, and interactions with other machines and humans, all of which directly or indirectly comprise "the information system." The number of different failure modes for only one of these components can be intractable, and when taken over all components in combination, the number of different ways that system failure can occur becomes "effectively infinite." For instance, can you enumerate all of the failure modes of a human operator of a electrical power plant when 5 different warnings occur simultaneously? And which of these failure modes will exacerbate the situation, and which are harmless? Today, we build systems that can tolerate certain anomalies, but we often fail to determine which other anomalies cannot be tolerated. These other anomalies constitute the weakest links of the system, and we need better methods for assessing the strength of the more brittle parts of the system. To make matters worse, the weakest links can quickly vary with only the slightest deviation in how the system is configured and used.

The problem is further exacerbated by the complexity imposed by concepts such as object-oriented programming and Web-based development. These technologies engender the possibility of programs no longer being monolithic, but rather being thousands or millions of little parts distributed globally, executing whenever called, yet still being part of one or many systems. These distributed technologies create a scalability problem for virtually all software assessment metrics.

The greatest impetus for using component-based technology for rapid development is the resulting gain in productivity. Today, organizations are touting future product cycle times of 3-4 months that were formerly 12-18 months. To accomplish this requires enormous success with reuse and componentization of fundamental objects that get reused in almost all developments. To make this happen gracefully, a development manager must be confident that fundamental objects constructed elsewhere are compatible and reliable for his or her application. The industry standard is that 50% of bugs are detected after component integration, not during component development and testing. Hence technologies for decreasing the problems of component integration are imperative.

In Leveson's book, "Safeware" [1], on pg. 36, she acknowledges the necessary role of interfaces in complex system design:

"One way to deal with complexity is to break the complex object

into pieces or modules. For very large programs, separating the program into modules can reduce individual component complexity. However, the large number of interfaces created introduce uncontrollable complexity into the design: The more small components there are, the more complex the interface becomes. Errors occur because the human mind is unable to comprehend the many conditions that can arise through interactions of these components. An interface between two programs is comprised of all the assumptions that the programs make about each other... When changes are made, the entire structure collapses.”

Later on pg. 410 of [1], Leveson succinctly describes the importance of thwarting the propagation of failures through the many interfaces that complex systems employ:

“A tightly coupled system is highly interdependent: Each part is linked to many other parts, so that a failure or unplanned behaviour in one can rapidly affect the status of others. A malfunctioning part in a tightly coupled system cannot be easily isolated, either because there is insufficient time to close it off or because its failures affects too many other parts, even if the failure does not happen quickly. Accidents in tightly coupled systems are a result of unplanned interactions. These interactions can cause a domino effects that eventually lead to a hazardous system state. Coupling exacerbates these problems because of the increased number of interfaces and potential interactions: Small failures can propagate unexpectedly.”

Recognition of the fact that the quality of today’s systems is highly dependent on tolerant interfaces has led us to develop an appropriate fault-injection methodology for software interfaces. The purpose of such a method is to assess the strength (failure tolerance) of the interfaces with respect to component failures and problems that enter in the system from external sources. Our method can be applied to source code interfaces or interfaces between executable components; however the thrust of this research pertains to interfaces between executable components. Interfaces are the mechanisms by which information is passed between components during processing, with the *final interface* being the mechanism through which the output passes. Our approach observes how interfaces behave when we intentionally corrupt data passing through them; the tolerance of an interface under simulated stress gives valuable information about the likelihood that the interface will produce acceptable results when the stress is real (i.e., after release).

Software components can be combined in series and parallel just as mechanical systems are. The most common way to combine components, whether physical or software, is to connect them in a series. When physical components are connected in a series, the quality of the whole is less than that of the worst component. However, for software, this may not be true, as another faulty component may actually make the system more reliable (by cancelling the effect of the other fault). Parallel component construction of physical systems is employed as a mechanism for increasing overall quality through redundancy. Once again, when this paradigm is applied to software, the expected increase in quality through redundancy may not occur. Hence the quality-based argu-

ments for designing from components that are used for engineering physical systems do not necessarily hold for software systems.

Because software components in series or parallel are not analogous to hardware components in series or parallel, it is prudent to assume that hardware engineering reliability theories are not applicable to component-based failure tolerance models. Our approach side-steps the series and parallel concerns, and instead focuses on the quality of a system’s interfaces.¹ By not looking at failure tolerance solely as a dependability problem, we are able to *explicitly* ignore how the components are interconnected, and place our attention on the interfaces that do the connecting. The results of our analysis technique (which will execute the code to give the information that we want to collect) are *implicitly* dependent on how the components are interconnected.

2.1 How the Technique Works

Our technique employs a modification of the fault-injection methods described for measuring software attributes such as software testability, safety, and vulnerability in [4, 3, 2]. Unlike the fault-injection employed in these applications, the new method described in this paper does not require access to component source-code. A major advantage concentrating on fault-injection on interfaces is that the analysis requires considerably less computing than was required when fault-injection was embodied in software instrumentation that is injected into the source code.

In this technique, there are two key events whose probabilities are *indirectly* measured after fault-injection is performed: “propagation across” and “propagation from.” Propagation across refers to the probability that an anomalous event that is input to a component will still be observable at the exit interface of the component, i.e., if the component gets bad input, will the component produce bad output? We need to know this information before we can assess how the failure of one component may affect another.

“Propagation across” a component will be quantified by first applying “perturbation functions” to interfaces feeding components their input data and then sniffing component exit interfaces to see whether the corruption propagated.² This process will be repeated n times, and the frequency of times that sniffing detects propagation will be used as the estimate of the probability. *Perturbation functions* are the fault-injection instrumentation mechanisms that we use to force incoming interface data to be corrupted. Assessing the likelihood of “propagation from” occurring will be harder, particularly if we are talking about “propagation from” an executable component. “Propagation from” is the case where something inside the component fails that then forces the component itself fails. If a particular component is very unreliable, then it has very high “propagation from.” “Propagation from” is very straightforward to assess via our previous fault-injection commercialized methods when access to source code is a non-issue [5]. But if source is not available, then fault-injection methods for executable components (that allow for corruptions to be injected via

¹Global parameters are considered as a part of the interface in our model.

²We are not only concerned with whether the exact same corruption that we injected propagated, but instead with whether any information coming out of the component is corrupt. After all, the original corruption may have caused a completely different corruption, which if observable at the exit point, signals that propagation across the component occurred.

instrumentation that is written in the executable language) will be necessary. Currently, we know of no ability to measure “propagation from” in this case.

2.2 flipbit

Information passed between interfaces is simply millions of 0s and 1s. To corrupt this information, we use a software procedure that inverts bits, `flipbit`. *flipbit* and its many derivatives (that cannot all be detailed here) are the perturbation functions that we employ.

The first argument to function `flipbit` is the original value that we wish to corrupt, and the second argument is the bit to be flipped (we assume little-endian notation). The function `flipbit` is then written in C as follows and linked with the executable. NOTE: The `^` represents the XOR operation in C and the `<<` operator represents a SHIFT-LEFT of `y` positions. Also, `~` represents the negation operator.

```
void flipBit(int *var, int y)
{
    *var = *var ^ (1 << y);
}
```

`flipBit` can be used to model various kinds of program state corruptions, including:

- n Random bits flipped, ($n \geq 1$).

```
void flipNbits(int *var, int n)
{
    int bits = 0;
    int bitPos = 1;
    int i,j,k;
    int xbit;

    for (i = 0; i < n; i++)
    {
        bits |= bitPos;
        bitPos <<= 1;
    }

    for (j = 0; j < sizeof(int) * 8; j++)
    {
        xbit = lrand48()

        if (((!(bits & (1 << xbit))) !=
            (!(bits & (1 << j))))
        {
            flipBit(&bits, xbit);
            flipBit(&bits, j);
        }
    }
}
```

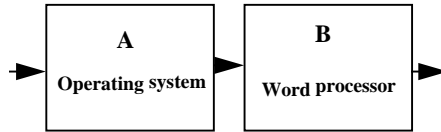


Figure 1: System S made of two COTS components in a series

```

}

for (k = 0; k < sizeof(int) * 8; k++)
  if (bits & (1 << k))
    flipBit(var, k);
}

```

2.3 Executable Code Fault-injection

Assessing the impact of components for which very little may be known is a conundrum. The expense of customized, one-of-a-kind systems has become so prohibitive that the practice of depending on someone else's code is ubiquitous. And third party code will almost certainly be delivered in a format that precludes any other party from seeing it. Without seeing such code, how can its impact be adequately assessed before it is added to a system?

To explain our approach to this problem, consider the system in Figure 1; Figure 1 represents a legacy system, S , comprised of two COTS components in a series. To implement the information-flow shown in Figure 1, suppose that there exists a main program that is composed of a Lisp-like series of calls where A returns a result to B.

```

Program
  B(A);
End.

```

Suppose however that a new executable component, D, is needed in S between components A and B, as shown in Figure 2. S is now:

```

Program
  B(D(A));
End.

```

We want to know whether D will seamlessly integrate into S . To know this, we will find answers to the following questions:

Question 1 If D is faulty, will its incorrectness affect the functionality of S ?

Question 2 If D is correct, but the interface between D and A is faulty, will that affect the functionality of S ?, and

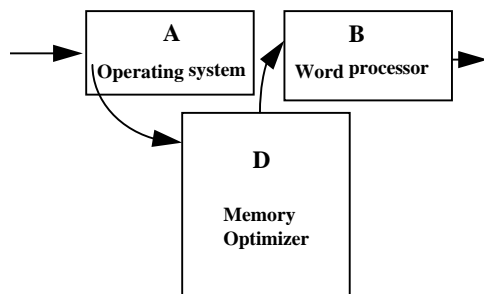


Figure 2: **Integrating Component D into S**

Question 3 If D is correct, but the interface between D and B is faulty, will that affect the functionality of S?

Question 4 Might D exacerbate any problems in A or B that have so far not noticeably degraded S's functionality?

To answer Question 1, we will inject artificial faults (representing possible faults in D) into the out-going parameters from D, by $B(\text{fault-inject}_1(D(A)))$. To answer Questions 2 and 3, fault-injection will simulate incorrect interfaces between D and A and also between D and B, which will include problems such as incorrect parameter orderings and incorrect parameters, via $B(D(\text{fault-inject}(A)))$ and $B(\text{fault-inject}_3(D(A)))$. Note that the fault injection functions which answer Question 1 and Question 3 differ in which parameters are injected. To answer Question 4, we can try combinations of events, such as $B(\text{fault-inject}(D(\text{fault-inject}(A))))$. Once this fault-injection has been performed, we gain insight into how likely it is that our modified S will continue to perform satisfactorily after the addition of component D.

2.3.1 UNIX Examples

This section discusses four examples of how applying fault-injection perturbations to the output of operating system calls can measure the impact of propagation across the calling programs. Software with embedded UNIX system commands could encounter problems, especially if it uses the output returned by the system commands in subsequent statements in the program. The output from a system call could get corrupted for a variety of reasons. Bugs in system calls have been well-documented and could cause such calls to return unexpected output [6]. Corruptions can occur within the stack of the program if memory resources have been exhausted. Physical failures in the computer could be responsible for noisy returned values as well. These are all examples of the impact that corruptions in the return values of calls to the operating system could potentially have on the calling application programs.

As a first example, the usage of the `vmstat` command is explored. A program may contain a `vmstat` system call to determine the amount of real and virtual memory currently available. Such a program may attempt to allocate

a large block of memory on condition that a sufficient amount of memory is indeed available. If the output from the `vmstat` command is perturbed numerically, a value indicating a larger amount of available memory than currently exists could potentially be returned to this program. The result could be disastrous for the program, which would fail to successfully allocate the needed amount of memory.

A second example involves the usage of the `mkdir` system command, which attempts to create a directory with a given name if possible. The `mkdir` command is representative of a large number of UNIX commands that output only a newline character if successful, and an appropriate error message if unsuccessful. For the `mkdir` command, error messages begin with the string “`mkdir:.`” A program that has not been carefully designed might test the resulting output assuming that output can be returned in only these forms. For instance, it could test whether the output from a `mkdir` call begins with the “`mkdir`” string, and determine whether the attempt to create a new directory failed or not based on the result. A corruption of a single character in the “`mkdir`” string that begins a resulting output error message would be misinterpreted as an indication that the `mkdir` command succeeded. On the other hand, a test based on the expectation that all successes will be indicated with a single newline character would encounter similar problems for the same class of perturbation. A single character perturbation causing a newline character to be at the beginning of a `mkdir` error message would result in the incorrect determination that the `mkdir` command succeeded, when in fact it failed.

A program that uses the `printenv` system command to determine an environment variable value would be susceptible to corruptions to the returned output. The result from a `printenv` command might be the path name for a directory in which to locate certain configuration files for that program. If the `printenv` output was perturbed such that extraneous characters appeared at the end of the returned string, the returned directory path name would be invalid.

Another case involves the usage of the `find` system command. Imagine a program that uses the `find` command to determine the full path name of a file it wishes to delete. If the file is located, and its full path name has been corrupted to the name of another file that exists, the wrong file would be deleted by the program. For example, a file called `critical_program.OLD` is to be located using `find` and then deleted. Suppose that it is located in the file system, having the path name `/home/apps/critical_program.OLD`. If the output from the `find` becomes corrupted in such a way that it is truncated to the path name `/home/apps/critical_program`, the program will attempt to delete the file named `critical_program` in the same directory. In the event that `critical_program` exists, the program will inadvertently delete it.

A commercial fault-injection tool exists that allows the user to define functions that perturb data values during execution of the program. The tool can be applied to software that makes UNIX system calls in order to perturb the result of those calls in order to determine the impact that a failure of UNIX will have on the software. To do this, the source code can be instrumented such that the data states associated with the returned output from the system calls are perturbed. The tool supports hardware-fault perturbations, standard random numerical perturbations for both discrete (integer) and real (floating-point) values, and assorted character string perturbations.

3 Summary

Our approach is applicable to both source code and executable Commercial Off-The-Shelf (COTS) components. Today it is commonplace for system developers to face the possibility that many of the key components of their systems are “black boxes.” They have no control of the quality in these boxes. A related benefit of our approach is that it can *assess* the failure tolerance of legacy systems, which even if the source exists, may be so unreadable and poorly documented that it might as well be written in an executable format. This measurement methodology provides assessments of system-level failure tolerance for any system that contains software whose quality is suspect.

Once it is decided that a component will be integrated into an existing system, it is prudent to explore the aforementioned questions about the possible affects of the integration. Fault-injection applied to interfaces is applicable to these questions. The benefits of applying fault-injection to component interfaces are that it:

1. Provides a measure of system-level *component-to-component interface tolerance*,
2. Provides a plausible measure of the likelihood of common failure modes between parallel components (whose function is supposed to be identical) as well as a measure of component independence for components in series, and
3. Provides a measure of the impact of a new COTS component on the quality of a legacy system in which the new component is substituted or otherwise placed.

In summary, predictions of component-to-component interface propagation provide a portion of the information necessary to predict failure tolerance. Interfaces are the means by which components communicate; interfaces are the glue for today’s information systems that are object-based. Failure-tolerant interfaces are imperative if we are to build failure-tolerant information systems.

Contact Address

Voas may be reached at RST Corporation, Suite 250, 21515 Ridgetop Circle, Sterling, VA 20166, USA, phone: (703) 404-9293, jmvoas@rstcorp.com.

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] N.G. LEVESON. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

- [2] J. VOAS, G. MCGRAW, A. GHOSH, F. CHARRON, AND K. MILLER. Defining an adaptive software security metric from a dynamic software failure tolerance measure. In *Proc. of Ninth Annual Conference on Computer Assurance*, National Institute of Standards and Technology, Gaithersburg, MD, June 1996.
- [3] J. VOAS AND K. MILLER. Dynamic testability analysis for assessing fault tolerance. *High Integrity Systems Journal*, 1(2):171–178, 1994.
- [4] J. VOAS AND K. MILLER. Software Testability: The New Verification. *IEEE Software*, 12(3):17–28, May 1995.
- [5] Reliable Software Technologies Corporation. *PiSCEs Software Analysis Toolkit (R) User's Manual*, 1996. Version 2.0.
- [6] B. P. MILLER, L. FREDRIKSON, AND B. SO. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, December 1990.