

Building Software Recovery Assertions from a Fault Injection-based Propagation Analysis

Jeffrey Voas

Reliable Software Technologies
Suite 250, 21515 Ridgetop Circle
Sterling, VA 20166

phone: (703) 404-9293 email: jmvoas@rstcorp.com

Abstract

We have investigated a fault injection-based technique for undermining the ability of software components to produce undesirable outputs into the state of the system. Undesirable outputs are any class of outputs that a component must not release into the state of the system given its current environment. Software components are said to be “failure-tolerant” if they release desirable outputs regardless of the programmer faults, potential malicious input data directed against the component, and other non-malicious but corrupted input data. Our technology assesses the failure tolerance of software components after simulated program state corruptions are injected into the components as they execute. Based on the types of outputs that result from fault injection, our technique knows where “recovery assertions” (which act somewhat like antibodies do in an organism) should be injected into software components to ensure desirable system outputs; the second part of our approach then suggests what the assertions should be.

1 Introduction

Today’s information systems include distributed hardware and software components, as well as interactions with other machines and humans. The number of different failure modes for just one of these entities can be intractably large. When all components are considered in combination, the number of different ways that a system failure can originate becomes effectively infinite. For example, try to list all of the failure modes for a human operator of a nuclear power plant when 5 different warnings occur simultaneously. We rarely know where the most unreliable component or subsystem is; it could be the human, the hardware, some component of the software, or some combination of these. Furthermore,

the most problematic component or subsystem can change with the slightest deviation in the way a system is employed. A total solution to these problems requires omnipotence, since we can never know all combinations of subsystem failure-modes.

So the problem is essentially twofold: (1) deciding where to place recovery mechanisms, and (2) building those recovery mechanisms. Software recovery techniques have been around for years can generally be subdivided into two categories: *backward recovery* and *forward recovery*. Backward recovery techniques return the software to a previous state that is hopefully not erroneous and continues computing using an alternative piece of software. Forward recovery attempts to repair the damaged state without rolling back the state of the program [3]. This paper proposes a new forward recovery heuristic, whereas other researchers have addressed the issue of using fault injection to test recovery code [1].

Our heuristic does not assume knowledge about the different ways in which the subsystems of a system can fail although that would further improve our results. Our methodology injects corrupt information into the states of an executing software component, and from doing, the methodology employs information typically lost after fault injection analysis to then build forward recovery mechanisms that counter the effects of that and other corrupt information. This is what differentiates our approach from other forward and backward recovery schemes [2, 3]. We do not claim that this approach will thwart *all* forms of corrupted information. This methodology only improves the prospects of satisfactory system operation by forcing software components to overcome anomalies during execution.¹

¹Here, “overcome” means producing output states that the rest of the system can tolerate, and an *anomaly* is corrupt information that is either fed into a component as input or corrupt program states created by the component.

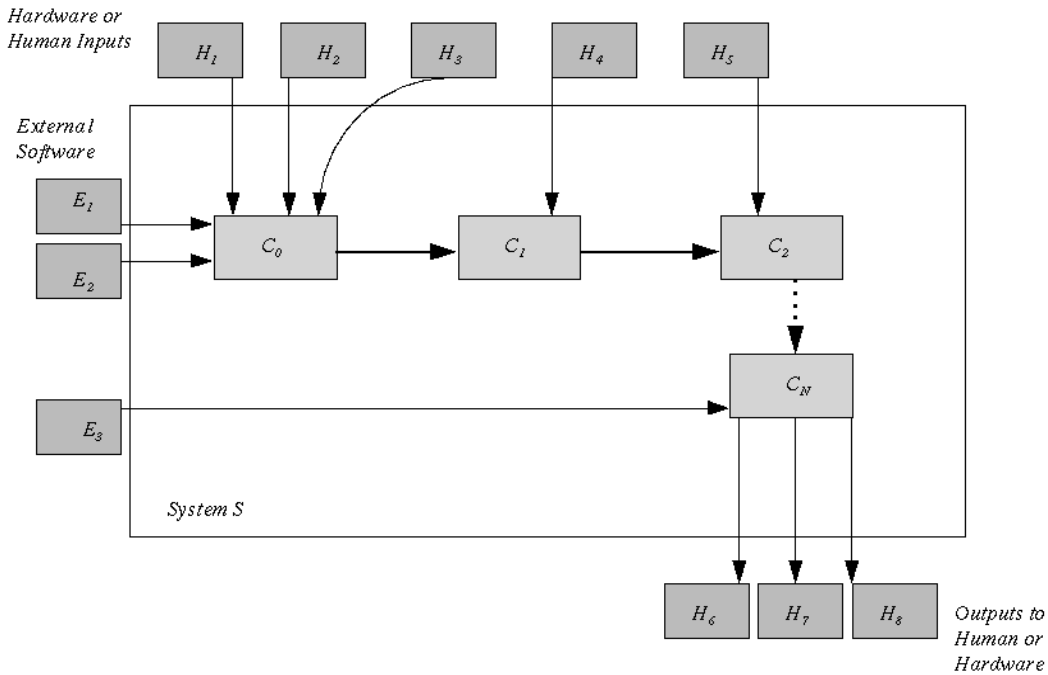


Figure 1: General Layout of a Component-based Information System.

2 Approach

Before we detail our approach, we will provide a simple example that illustrates the problem. Figure 1 shows a layout of an information system S comprised of $N+1$ components (C_0 through C_N). The complete environment in which S resides includes S , E_1 - E_3 and H_1 - H_8 . S receives a variety of incoming inputs, from hardware devices or human operators (shown as H_1 through H_5), and it also receives input signals from three external software components: E_1 , E_2 , and E_3 (which could even include Commercial-Off-The-Shelf software components). S sends three output signals to H_6 , H_7 , and H_8 .

Let's concentrate solely now on C_1 . (In practice, the following process will be done for each component.) C_1 gets its inputs from H_4 and C_0 , and sends its output to C_2 . From a traditional fault tolerance standpoint, the goal is to ensure that *any* anomalies that might exist in the internal program states of C_1 do not cause the output of C_1 to be such that H_6 , H_7 , or H_8 receive faulty data. Our goal is slightly different: we will gladly allow C_1 to output faulty data so long as the data does not lead to H_6 , H_7 , or H_8 receiving unsafe or security-compromising data. Anomalies could arise in the program states of C_1 from a variety

of sources, including faulty data from H_4 , faulty output from C_0 , or defects within C_1 . If our approach observes that such events could cause the output of C_1 to be such that is likely to impact S 's output in an undesirable manner, then C_1 is a candidate for recovery heuristics. If C_1 cannot be modified to be more tolerant of such anomalies, then improving the *reliability* of C_0 , C_1 , and H_4 must be considered in order to ensure that H_6 , H_7 , and H_8 do not receive undesirable outputs (unsafe or vulnerable). The main goal of this technique is to “force” software components to be robust with respect to their sources of potential problems (such that the components do not output events that could cause subsequent system-level catastrophes).

A secondary objective is to develop recovery heuristics that are cost-effective and can be applied to real-time software. This objective was motivated by a commercial safety-critical application. After using fault injection to analyze the system's tolerance to corrupted states, the developer found that 30 recovery mechanisms were needed to ensure that particular state problems did not manifest. This situation was exacerbated by the real-time constraints that allowed for a maximum of 3 to 5 assertions to be added into the code. As a result, it was critical that the

few recovery assertions perform multiple tasks, i.e., one assertion needed to do the work of several. The problem was determining how to do so.

The approach is four-phased. First, a fault-injection technique called *extended propagation analysis* (EPA) [4, 6] is performed on the software to determine where data-state corruptions can originate that propagate into unacceptable outputs. The second phase, called *static error flow analysis* (SEFA), analyzes how different program state spaces are dependent on one another (with respect to the different source-code locations that are associated with them). By combining the results of the first two phases into a single metric, we can identify regions in the code where we can effectively thwart anomaly propagation from multiple source code locations via appropriate recovery mechanisms. The third and fourth phases are responsible for building these “appropriate recovery mechanisms.” The third phase, called *program state classification* (PSC), develops a database of program states to determine whether or not they will likely lead to unacceptable output. The fourth phase, called *recovery assertion injection* (RAI), places a *recovery assertion* (that is based on the classifier) at those places in the code identified from the first two phases as most likely to improve the fault tolerance of the component. Recovery assertions attempt to replace existing program states with other states in order to return executing components to states that will result in acceptable component outputs.

As an analogy, consider a complex chain of dominos. This chain corresponds to the dynamic execution of software. Chains of dominos branch out, sometimes reconverge, and result in “outputs” at the last dominos of each chain. Pushing over a domino so that it topples and starts a chain reaction is equivalent in our analogy to introducing an anomaly that propagates through the executing program. Just as sometimes felling one domino can result in many dominos falling, introducing a fault in a running program sometimes corrupts many program outputs in unacceptable ways. Our objective is to stop the last domino from falling. This can be accomplished by inserting immovable dominos throughout the chains. Realize, however, that inserting one enormous immovable domino before the last domino is not a realistic option since such a domino would have to essentially be an oracle. So we will need to determine where else in the chain to insert immovable dominos while at the same time minimizing the number of them.

2.1 Extended Propagation Analysis (EPA)

As shown in Figure 1, for a software component to be failure-tolerant, there are three major classes of pitfalls that the component must be able to protect against: faults in the component, the effects of failures in other software subsystems, and inputs from hardware that has failed.² At the component level, inputs from hardware and other software subsystems that has failed can be modeled as erroneous incoming data to the component.

Extended propagation analysis is a dynamic fault injection-based technique that instruments source code so that it can simulate all three classes of problems; this technique has already been published in [6], and we will use Algorithm 1 from that paper as the first algorithm of this approach. This algorithm allows us to assert (with confidence) what the net impact will be to output variables according to the failure classes (hardware and software) that are simulated by EPA. We can also estimate the probability that these failures (represented as corrupt data states by EPA) will result in catastrophic outputs from a component.

2.1.1 Defining $PRED_{C_i}$ with Respect to the Complete Information System

EPA requires definitions for what constitutes unacceptable component output from a component, C_i . This information can be: (1) backed out from a forward hazard analysis of the system, S , or (2) determined from C_i ’s specification, and is encoded into one logical predicate, $PRED_{C_i}$, for each component. This describes all types of output that the component is not to produce given certain contexts. If (1) is the manner by which $PRED_{C_i}$ is determined for C_i , then we are talking about two different forms of unacceptable component behavior: one with respect to S , and one with respect to C_i without regards for S . The definition for what constitutes undesirable system behavior will, in part, drive the definition for $PRED_{C_i}$. For example, in Figure 1, we can simulate anomalies exiting C_1 , and see if any of those anomalies force unacceptable outputs in H_6 , H_7 , or H_8 ; if they do, then we know something about what types of states we do not want to exit C_1 , and can build that information into $PRED_{C_1}$. One way to determine what events we do not want C_i to present to the system is to use static fault-tree analysis.

²Hardware failures are sometimes called “hardware faults.”

2.2 Static Error Flow Analysis (SEFA)

Following EPA, the second phase in our approach, Static Error Flow Analysis (SEFA), is applied to analyze data dependencies between the different locations in a component. SEFA can be thought of as a static slicing algorithm [7] in reverse. This phase is vital to reduce the number of recovery assertions needed. A *data space* is the set of all program states that can be created by a location l . In the domino analogy, the function of SEFA is to study the domino chains and find the fewest locations where a single barrier halts the most domino toppling. For example, in dominos, if many different lines of dominos merge into a single line, then placing a barrier in the merge line is more cost effective than placing a barrier in each incoming line.

SEFA begins by first hypothesizing an initial corruption in some variable a at some location l . Next, SEFA predicts *which* set of variables might be corrupted after executing some subsequent location w . SEFA creates sets of these variables, called *fault sets*, revealing dependencies between data spaces. Furthermore, SEFA assigns a metric for rating the domino chains in order to identify critical fault sets. The reason for performing SEFA is to prune the number of locations that are candidates for recovery assertions in order to boost the component’s fault tolerance within real-time constraints. SEFA is implemented by Algorithm 2:

Algorithm 2: Static Error Flow Analysis (SEFA)

1. For each l in C_i ,
 - (a) Set k to l ,
 - (b) Assume a fault in l causes the variable on the “left hand side” $\text{lhs}(l)$ to be corrupted (*initial corruption assumption*). Set $\mathbf{fault_set}_{l,k}$ ³ to the set containing $\text{lhs}(l)$. If l is a conditional expression, then $\text{lhs}(l)$ is obtained from the location pointed to by the program counter.
 - (c) For each location w that is *statically reachable from l* , *BUILD*

³In our notation, $\mathbf{fault_set}_{m,n}$, the first parameter m represents the location where the initial corruption supposedly occurred, and the second parameter n is the location just executed before the fault set is built.

($\mathbf{fault_set}_{l,w}$).

Algorithm BUILD ($\mathbf{fault_set}_{l,w}$):

- i. Set $\mathbf{fault_set}_{l,w}$ to $\mathbf{fault_set}_{l,\text{prior}(w)}$, where $\text{prior}(w)$ is any location that could be executed immediately before this iteration of w and $\text{prior}(w)$ does not equal null.
- ii. If the “right hand side” of w , $\text{rhs}(w)$, references a variable in $\mathbf{fault_set}_{l,\text{prior}(w)}$, then add $\text{lhs}(w)$ to $\mathbf{fault_set}_{l,w}$. If w is a conditional expression that references a variable in $\mathbf{fault_set}_{l,\text{prior}(w)}$, then add $\text{lhs}(w)$ referenced by the program counter to $\mathbf{fault_set}_{l,w}$ and repeat for all branches of the conditional expression.

The results of this algorithm provide static information needed by the Calculator concerning the data flow relationships between different locations and our static predictions as to “what” can corrupt “what else.” Given Algorithm 2, the fault set for location w , called π_w , is:

$$\pi_w = \bigcup_{\text{for each } k} \mathbf{fault_set}_{k,w}$$

This provides a prediction of the set of all variables that are likely to be corrupted after executing location w , assuming that some location executed before w caused an initial corruption.

We now wish to provide a single numerical score, s_w , for each location w in the code. s_w represents the degree of fault tolerance of the component if the program state created by w is corrupt:

$$s_w = \sum_{\Gamma} \psi_{a|C_i,Q}$$

where Γ represents each a in π_w where l is executed before w (The higher s_w is, the lower the predicted failure tolerance.) For those s_w ’s that are large, a recovery assertion should be inserted immediately after location w to force the variable on the $\text{lhs}(w)$ into a range that is unlikely to satisfy $PRED_{C_i}$. Let L represent the set of all w ’s that do so. A single recovery assertion of this sort can measurably improve the component’s ability to recover from anomalies originating from each location that contributed a member to π_w .

2.3 Recovery Assertions

Section 2.3 describes the last two phases in our technique. As already mentioned, after the first two phases of our approach are completed, we have completed the process of identifying where recovery assertions are needed. At this point, exception handlers could be employed instead of our type of recovery assertion. The purpose of the last two phases of the approach is to determine what repair should be recommended once it is detected that a component is in an undesirable state.

“Testing” assertions detect state anomalies. These assertions test current program states for out-of-range variables, for the presence or absence of certain relationships between variables and inputs, and for known corrupted states. When an assertion triggers (ignoring the possibility of false-positives), a corruption has been detected. Unlike testing assertions that usually only produce warnings, recovery assertions attempt to modify the state. Each recovery assertion is highly application-specific. Algorithms 1 and 2 above do *not* solve the problem of deriving recovery assertions, they merely identify locations in the code where they are *more likely* to be effective at repair.

Recovery is not foolproof! The difficulty is in knowing what is an acceptable component state with respect to the current system state. Recovery assertions are similar to classical pre- and post-conditions that determine correctness but are not formally derived from a specification. Like classical pre- and post-conditions, recovery assertions are subject to misuse.

Recovery assertions are similar in purpose to exception handlers. What is unique in our paper, however, is how we decide where to place the recovery assertions and what the recovery assertions should be. It would be plausible to only perform the first two phases of our approach and base exception handlers on the results of the first two phases.

There are two tasks performed by recovery assertions:

1. *Detection* of states that will likely lead to unacceptable events.
2. *Substitution* of acceptable states for *probably* dangerous states.

Our approach here will be to: (1) perform Algorithm 3 repeatedly at those locations whose s_w was large, and (2) implement Algorithm 4 (that creates and inserts the instrumentation of the recovery assertions into C_i). Because the number of assertions that we can feasibly expect to inject may be few, the threshold defining “whether a particular s_w was large” may

itself be quite large. So the number of recovery assertions that are deemed as feasible will directly determine where Algorithm 3 is applied.

Algorithm 3: Program State Classification (PSC)

1. Create an empty set, $UNACCEPT_w$.
2. Create an empty set, $ACCEPT_w$.
3. Randomly select an input x according to Q , and if C_i halts on x in a fixed period of time, find the corresponding $\mathcal{A}_{IC_i x}$ in $\alpha_{IC_i \Delta}$. Set \mathcal{Z} to $\mathcal{A}_{IC_i x}$.
4. Alter the sampled value of variable a found in \mathcal{Z} , and denote the new data state as \check{z} , and execute the succeeding code on \check{z} . If by chance w is executed more than once because of the value we placed into \check{z} , alter a in each state that is produced by w .
5. If C_i 's output satisfies $PRED_{C_i}$, add \check{z} to $UNACCEPT_w$. If C_i 's output does not satisfy $PRED_{C_i}$, add \check{z} to $ACCEPT_w$.
6. Repeat steps 3-5 for as many x 's as resources allow.

Algorithm 3 classifies states for detection and recovery purposes. Because recovery at w repairs at places other than w , the process has additive side-benefits. Algorithm 3 builds up two sets of states for location w : (1) one set made up of those states that have been observed to lead to unacceptable output, and (2) another set of corrupted states that did not lead to unacceptable output. Once we have compiled sets of program states that are acceptable (in terms of the outputs they produce), we have a pool from which to choose any substitution that may be necessary during operation.

The recovery assertions that are developed by PSC are a function of: (1) Q , and (2) S 's output states that are defined as undesirable. This begs an interesting question: “What value, if any, do the recovery assertions added to C_i have when C_i is placed into another system S' which might have different definitions for undependable system behavior?” If $PRED_{C_i}$ is defined specifically for S , then it is possible that this four-phase process would need to be reperformed for each different S . If, however, $PRED_{C_i}$ is defined according to C_i 's specification, then the recovery assertions should be valid for each S that C_i is embedded in (if C_i is the correct component for S).

For each location whose s_w is found to be unacceptably high, we will augment the code of C_i with the appropriate instrumentation: states that are known to cause problems with states that are known to not cause problems.

Algorithm 4: Recovery Assertion Instrumentor (RAI)

Immediately after each execution of location w , insert recovery code that performs the following requirement: Test to see if the current state is in $ACCEPT_w$; if so, then no action is required, else if the current state is in $UNACCEPT_w$, then swap out the current state for some other member in $ACCEPT_w$ if and only if it is determined that the current state's system input value is "similar enough" to the replacement state's system input value.

This algorithm is a high-level description of a recovery assertion. The key problem here is in determining when a state is similar enough is a system-state-specific task. Determining whether a substitution will not worsen a situation is dependent on the state of S . For some components, an assertion may be adequate for making this decision. In other components, an examination of the states in $UNACCEPT_w$ may reveal a pattern in the unsafe states that can be captured and codified in a classifier. In our current studies, the transformation of $PRED_{C_i}$ and $UNACCEPT_w$ data into some procedure that determines whether or not states are similar to what we already know about is a manual process.

3 Summary

We have described a fault injection-based approach to cost-guided software recovery, however the idea of software recovery has been around for decades [2]. Our approach presents several advantages:

1. Our approach gets the needed information for where and how to recover from the observations made during fault-injection analysis.
2. Our approach can be combined with other forward recovery techniques that prevent single-cycle errors.
3. Our approach recommends recovery mechanisms only at the fewest points possible. Since recovery is expensive, this is valuable.

4. The user can objectively observe and measure any increase in failure tolerance that the injected recovery assertions provide.

Similarly, there are weaknesses in our approach, mainly the reliance on access to definitions of undesirable system behavior in $PRED_{C_i}$ as well as the possibility that a state substitution will occur that worsens the situation. At this point, we do not have solutions to these problems.

Acknowledgements

The approach described in this paper was first presented at [5]. 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.M. BIEMAN, D. DREILINGER, AND L. LIN. Using fault injection to increase software test coverage. In *Proc. of International Symposium on Software Reliability Engineering*, pages 166–174, White Plains, NY, October 1996.
- [2] B. W. JOHNSON. *Design and Analysis of Fault Tolerant Digital Systems*. Addison-Wesley, 1989.
- [3] N.G. LEVESON. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.
- [4] J. VOAS AND K. MILLER. Dynamic testability analysis for assessing fault tolerance. *High Integrity Systems Journal*, 1(2):171–178, 1994.
- [5] J. VOAS AND K. MILLER. The Avalanche Paradigm: An Experimental Software Programming Technique for Improving Fault-tolerance. In *Proc. of the IEEE Int'l. Symp. and Workshop on Eng. of Computer-based Systems*, pages 142–147, Friedrichshafen, Germany, March 1996.
- [6] J. VOAS, F. CHARRON, G. MCGRAW, K. MILLER, AND M. FRIEDMAN. Predicting How Badly 'Good' Software can Behave. *IEEE Software*, July 1997.
- [7] M. WEISER. Programmers use slices when debugging. *CACM*, 25(7):446–452, July 1982.