Better Reliability Verification in Open-Source Software Using Efficient Test Cases

Patrick Pape, Mississippi State University
Drew Hamilton, Mississippi State University

Abstract. With the increasing popularity of open-source solutions in projects across varying domains and levels of dependability requirements, there is a need for a way to efficiently bring open-source software to a level that passes reliability verification testing before being integrated into a pre-existing system. The primary issues with integrating open-source software into a system is that more often than not the developmental methods cannot be verified and the software is already in a post-release version. So, how do you retain the benefits of utilizing open-source solutions to problems while bringing the open-source software to a reliable operational level that meets specifications for your project? In this article, we will discuss a method for efficiently locating key areas for the placement of error handling in order to increase fault tolerance and for drastically reducing the number of tests necessary to verify that the open-source software to be integrated meets specifications.

Introduction

Using open-source software in lieu of consumer and government off-the-shelf options for adding modular functionality to software projects or as a foundation for starting new software projects is a rapidly trending upward. [1] In our experience the government prefers an open-source software option when feasible and when robustness and security requirements of the project allow. There is abundant open-source software that performs operations across most applications. Primary drawbacks to utilizing open-source software include: no guarantee of dependability, unreliable development methods and the costs of integrating software in the late stages of development. Despite these drawbacks, open-source software is still seeing a large increase in private sector usage. [2] Open-source software can be used in many cases to reduce both the time and cost of adding new functionality or starting the development of a new project. The method discussed here will detail how to retain these cost and time savings while overcoming the reliability limitations of open-source software in systems with strict specifications for verification testing.

The inspiration for this work comes from academic research clients interested in using open-source solutions in order to save time and money in spite of software reliability concerns. The primary issue from a development point of view was trying to look at post-release software and implementing some methods for increasing the reliability of the code and performing reliability verification. One key example was an open-source mission control software for a programmable UAV. The mission control module was a mission-critical portion of the system and had strict specifications for dependability measures. The time it would take to test the system and bring it up to specifications was more than the time it would take to find alternate software that performed the same task. This dismissal of open-source software as a valid alternative to high specification systems lead to researching a way to make the method more efficient and the code more reliable.

The problem with most current solutions to the problem of making the verification process for software more efficient is that there is a lack of consideration for dealing with software only in the late stages of development and in post-release. With open-source software, unless you are involved in the development at some early stage, you are likely attempting to incorporate it into an existing system for some added functionality. This means that the solutions which focus on working on verification through each developmental iteration are not of use. A different approach must be taken in order to bring the software up to the desired level of reliable operation with acceptable overhead. Adding error handling in the late stages of development can add large overhead. The testing process discussed in this article will detail a method for making this last stage verification and reliability enhancement process more efficient.

Importance of Variables

The method utilizes static and dynamic analysis of the software to reduce the total number of tests needed to verify the software and to determine the key locations for placing error handling in order to bring the software up to the required level of reliable operation. The method focuses on utilizing the relative use of the open-source software to be integrated into an existing system to expand functionality. A recently developed metric known as importance [3] is used as a baseline for determining the priority of the variables in the software that are the highest priority for error handling. The drawbacks to this approach are a lack of cross-module measurability, meaning that each measure of importance is only relative to the other variables in a single module. The approach described here adapts the metric to establish the importance of variables across an entire system, including a large number of modules and functions.

The importance metric basically works by determining the failure rate of a variable, \( f \), its spatial impact and its temporal impact, written as:

\[
I_{v,C} = 1 + (1 - f)^m \times (\sigma_{v,C} + \sigma_{\text{max}} + \tau_{v,C} + \tau_{\text{max}})^n
\]

where \( m \) and \( n \) are coefficients that can be modified to place more or less emphasis on either the failure rate of a variable or its impact on the system. [3] Failure rate in this scenario is the...
frequency that a fault injected into a variable lead to a full-blown failure in the system. Spatial impact is defined as the diameter of the area in the code that is affected by the failure, written as:

$$\sigma_{v,C} = \max\{\sigma_{v,C}\}, \forall r,$$

[3] and the temporal impact is defined as the duration that a program is affected by the failure, written as:

$$\tau_{v,C} = \max\{\tau_{v,C}\}, \forall r.$$

[3]. Basically, these equations state that the impacts measure the diameter of the affected area and amount of time the program state remains affected, when a variable v in component C is corrupted. A higher spatial impact indicates the difficulty of recovering from the corruption and a higher temporal impact indicates a higher chance for the program to fail.

Software Faults

The capability of utilizing an accurate measurement of the importance of a variable in a system relies on the concept of relating variables in different functions in different modules of a system to each other. The common trends of software faults and failure data from real-world case studies is explored in [4] and provides focused discussed on the localization of faults which can lead to varying types of software failures and the distribution of failures in a system. The conclusion of this study was that the primary types of faults include: requirement faults, coding faults and data problems. Of importance to the issue at hand is the conclusion that the trends of software failures are intrinsic characteristics of software faults and not specific to the individual project. This coincides with the belief that it is possible to create a metric to measuring the relative likelihood and impact of faults across any system.

Detecting and Correcting Faults

There are numerous ways to increase the reliability of a system using data flow analysis, including check-pointing [5], information flow relations [6], and other techniques. Check-pointing involves looking at the code at the instruction level and splitting it between protected and unprotected sections. In the protected code the data values are replicated and are compared at branch instructions to check for discrepancies. This concept is utilized with respect to the error handling placed into the code after the relative importance metric is calculated to ensure accurate reading and writing of variables. This error handling is currently in the form of a wrapper-based function call for each read and write of variables dependent on the level of importance. [7] The most important variables are triplicated and less important variables are duplicated during writes. This means that when the variable is being read during a program statement, the wrapper-function will check what the most common value is and return that. This allows for some data corruption without jeopardizing the values of the variable completely. Information flow and state flow analysis [8] can be used to detect errors in variables and program statements that cause undesirable actions and states in the software.

Fault-Injection Framework

There are a number of fault-injection frameworks that could be utilized depending on the application and structure of the software that is to be tested, including: PROPANE [9], MESSALINE [10] or FIAT [10]. PROPANE is an environment that supports fault-injection through mutation of source code and data errors by manipulating variables. MESSALINE [10] and FIAT [11] are used as sources of information as far as design considerations for the framework, but like PROPANE were proposed years ago and have since become less prominent solutions to new fault-injection problems. Current fault-injection frameworks are generally limited in usefulness based on the structure of the software system being tested and rely on being a part of the V&V testing process before deploying the software. These frameworks are useful for investigating the effects of individual faults on a system and identifying potentially vulnerable locations in code, but the results are not context sensitive.

The base of the method is the fault-injection framework that was written specifically to work for this particular project. The framework utilizes two models for injecting faults. The global model constrains the occurrence of the faults to dependability measurements and assumes any variable in the system could be affected by the data fault. The local model states that the types of faults to occur in the system will be transient data value faults. This means that the faults can occur at any time and any place in the software and will not remain in the program after execution stops. The framework can be split into three main components: injectors, probes, and environment simulator. The injections are done manually by inserting code into the source, depending on the type of fault different inputs are needed. For boundary testing, the desired injection value is required, otherwise the injection will target a random bit in the memory space for the variable and will flip the bit. The probe component is implemented through inserting code into the source that records the value in certain variables at different points throughout the execution of the code. The environment simulator works by emulating the existing system and controlling the execution of the target software during the intended test cases. This is done in order to get accurate results for the specific use cases of the open-source software during the testing phase.

Case Study – Mp3gain

Testing is based on an open-source program called Mp3gain which can be used to normalize volume and other audio modifications to mp3 files. Mp3gain is highly modular and was developed primarily by a single developer, a common occurrence with open-source software. The goal of the experiment was to utilize Mp3gain with several real use cases. Three test cases were selected: scan album for maximum gain adjustment, normalize volume across all tracks, and undo all changes to album. The input to the experiment was an album of 25 tracks, where faults were injected 25 times in each test at 25 equally spaced points. So, there was a single fault injection per track analysis. For each fault injection test for a variable, the program was run 100 times, meaning that each use case was run 100 times with 25 input tracks.

The first round of testing obtains the worst case impacts and determine the failure rate. The second round of testing used the probability of execution found in stage one to create a number
Defining a Critical Path

The first step in the method is to determine the critical path, or the key flows of data throughout the source code. This requires that we instrument the code in order to determine which of the modules, and the functions within, are most important to the intended use case of the system. The idea is to reduce the number of unnecessary tests cases at the beginning of the testing process. The frequency that each function in each module is called during the running of the use cases is measured. This frequency value is then used along with the worst case lines of code in a function to determine the effective lines of code for each function written as:

\[ \text{ELoC}_f = \text{LoC}_{wc} \times F_{cf} \]

[12]. This measures the total number of lines of code that will be run during the execution of the program for that function. For the sake of brevity, this particular table is left to the reader to examine in [12] as it includes a large number of entries.

The next step in this stage is data flow analysis on the source code to generate the caller and call graphs for each function. This is done in order to see which functions and modules are communicating with each other, so that we can track where the data is going. This is important because it helps to gauge the propagation factor of each of the functions in the code. The propagation factor is the call depth of a function divided by the maximum call depth of any of the functions, written as:

\[ p = F_{cd} / \text{Max}_{cd} \]

[12]. Functions that have a high number of caller functions are more likely to spread corrupted data. Once these functions are given a value for priority, the priority of the modules are determined. The caller graph for the first critical path of the case study can be seen in Figure 2. The figure shows the most critical flow of data throughout the target software, where the focus of the placement of error handling will be. Each box represents a different module in the system.

A functions’ priority, written as:

\[ \text{Priority} = P_c \times P_e \times P_f \]

[12], is calculated based on the probability that a single bit fault will occur in memory related to a function, probability that the code corrupted by this fault is run in the current execution of the program, and the propagation factor. The probability of execution is written as:

\[ P_e = \text{ELoC}_f / \text{TELoC} \]

[12], meaning the effective lines of code in the function divided by the total effective lines of code across all functions. The module priority is the summation of the priority of the functions that compose the module. A more detailed look at each stage of the method can be found in [12].

Identifying Locations for Error Handling

Once the critical path throughout the code is found, we must determine the relevant variables within the critical path to be tested. All the variables from high priority modules on the path are added. For each of these variables, a dynamic program slice is done for both the caller and called graph to get a clear measure of all the variables that are affected and affect the variable that is being analyzed. Variables with the highest relative importance ranking are located in this critical path. The output of this step is the list of all the relevant variables that gets passed to the next stage of instrumentation for fault injection testing.

Fault injection test cases are run for each relevant variable found in the critical path to obtain metrics to measure relative importance. These include: spatial impact, temporal impact, relative failure rate, and failure rate, for validation. The importance metric was taken from its generic form:

\[ I_{v,C} = G[K(\sigma_{v,C})L(\tau_{v,C})] \]

in [3] and is used with the failure rate and impact of the variable to determine the importance of a variable within its own function. This equation is modified to give the importance of a variable relative to the intended use cases of the target software:

\[ RI_{v,C} = 1 / (1 - f_i)^n \times (\sigma_{v,C} / \sigma_{max} + \tau_{v,C} / \tau_{max})^m \]

[12] and compared to the local importance.

The results for the top threshold of variables in the case study can be seen in Table 1. This shows the top fifteen percent of variables in terms of relative importance across variables in the system. Note that given how the program handled albums, the
faults that manifested as failures would either cause failure for all twenty-five tracks or just one, leading to the similar temporal impact values. Another interesting note is how the curframe variable was ranked in the main module. The relative failure rate was incredibly low, but it was still ranked the sixth most important variable. This was because the fallout in the system from having this value corrupted was bad enough to outweigh the miniscule chance that the variable be read after having been corrupted.

**Wrapper-Based Error Handling**

An efficient and verified error handling mechanism from the same paper as the original importance metric is used. There are two stages of wrapping used in this error handling. First, any time an important variable is written to, the wrapper function is called and copies of the value are stored in case the original variable value is corrupted. The second stage is during reads of an important variable. When an important variable is read, another wrapper function is called which uses a majority voting algorithm to return the correct value. Custom thresholds are determined based on acceptable levels of overhead for determining how many variables will be wrapped. This method of wrapper was shown to be lightweight, efficient and well tested in [11].

**Results**

The results of the case study for the method are reassuring. The method is able to effectively reduce the total number of tests needed for reliability verification by using a measured means of removing irrelevant and unnecessary tests and focusing on only the most important variables. Table 2 shows the results of the comparison between using the given method and just the local importance. The failed tests column indicates the total number of test that resulted in system failure that occurred and the relative failed tests indicate how many of these failed tests involved variables with a significant importance rating in the system. The results show the discrepancy between the total number of test cases and how many of those test cases would have a significant impact on the system.

Table 2 also shows a clear measurement of the number of failed tests found using a modified failure rate that is relative to the whole system local measurement of failure rate. Remember that the modified failure rate is determined utilizing the probability of execution of the corrupted data. Instead of the assumption that all data corruptions are equally likely to occur, by disregarding the probability of the data fault being executed, the modified failure rate takes this into account. The number of failures that are found with the method is significantly greater than with just local module analysis.
Table 1 - Fault injection results for variables for critical path one [12]

<table>
<thead>
<tr>
<th>Function</th>
<th>Variable</th>
<th>$\sigma_{v,C}$</th>
<th>$\tau_{v,C}$</th>
<th>Failure Rate</th>
<th>Relative Failure Rate</th>
<th>Relative Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>synth_1to1</td>
<td>bandPtr</td>
<td>3</td>
<td>25</td>
<td>0.0138666667</td>
<td>0.0088</td>
<td>1.7816802003</td>
</tr>
<tr>
<td>synth_1to1</td>
<td>pnt</td>
<td>3</td>
<td>25</td>
<td>0.0088</td>
<td>0.0056</td>
<td>1.769765878</td>
</tr>
<tr>
<td>do_layer3</td>
<td>pcm_point</td>
<td>3</td>
<td>25</td>
<td>0.0284</td>
<td>0.0002666667</td>
<td>1.7511506951</td>
</tr>
<tr>
<td>synth_1to1</td>
<td>buf</td>
<td>2</td>
<td>25</td>
<td>0.0321333333</td>
<td>0.0206666667</td>
<td>1.5640653421</td>
</tr>
<tr>
<td>synth_1to1</td>
<td>maxAmpOnly</td>
<td>2</td>
<td>25</td>
<td>0.0081333333</td>
<td>0.0052</td>
<td>1.5158384332</td>
</tr>
<tr>
<td>main</td>
<td>curframe</td>
<td>2</td>
<td>25</td>
<td>0.0188</td>
<td>0.0000001416</td>
<td>1.5</td>
</tr>
<tr>
<td>synth_1to1</td>
<td>window</td>
<td>1</td>
<td>25</td>
<td>0.0734666667</td>
<td>0.0472</td>
<td>1.3772426984</td>
</tr>
</tbody>
</table>

Table 2 - Number of failed tests comparison [12]

<table>
<thead>
<tr>
<th>Critical Path #</th>
<th>Failed Tests – Method</th>
<th>Failed Tests – Local</th>
<th>% difference</th>
<th>Relative Failed Tests – Local</th>
<th>Relative Failed Tests – Method</th>
<th>% difference Modified Failed Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP1</td>
<td>2135</td>
<td>3647</td>
<td>-70.82</td>
<td>110</td>
<td>1145</td>
<td>90.39</td>
</tr>
<tr>
<td>CP2</td>
<td>6601</td>
<td>6551</td>
<td>0.76</td>
<td>1110</td>
<td>1406</td>
<td>21.05</td>
</tr>
<tr>
<td>CP1/2</td>
<td>5398</td>
<td>9100</td>
<td>-68.58</td>
<td>1155</td>
<td>1724</td>
<td>33.01</td>
</tr>
</tbody>
</table>

Table 3: Method vs. Local Cost-Benefit Analysis [12]

<table>
<thead>
<tr>
<th>Mp3gain</th>
<th># of tests – method</th>
<th># of tests Local Cost-Benefit Analysis</th>
<th>% difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP1</td>
<td>47</td>
<td>551</td>
<td>91.47</td>
</tr>
<tr>
<td>CP1/2</td>
<td>97</td>
<td>551</td>
<td>82.40</td>
</tr>
</tbody>
</table>

Table 3 shows a reduced number of total tests required to identify the most important error handling locations in the open-source software. Given these initial results, the method appears to accurately locate variables with a high relative failure rate, in addition to reducing the total number of tests needed to complete this reliability verification process. Meaning that the method should decrease the time needed to verify the reliability of the target software when compared to local cost-benefit analysis, which requires that each of the variables in the system be tested in order to get an accurate reading on where to prioritize error handling mechanisms.

**Conclusion**

The method shows a satisfactory reduction in testing time and accurate identification of locations for error handling placement in open-source software components to be integrated into a pre-existing system. This serves as a step in the right direction for promoting a wider usage of open-source solutions in various domains. Though directed at open-source software, any post-release software with available source code could be tested using this method. The current shortcomings of similar solutions of needing to be utilized from the early stages of development and a lack of bias in the modular structure of the system do not apply for this method. The discussed method incorporates an understanding of system structure and dependability properties to give an insight into the relative operation of the system for intended use cases of the software. This allows the user of the method to focus their reliability verification testing only on the code that affects how they intend to utilize the code. The next step for this method is to further reduce the number of tests required to locate key variables for wrapping and to determine how the metrics used in the method can be expanded upon to be more accurate.
Dr. Patrick Pape is an Assistant Research Professor with the Distributed Analytics and Security Institute (DASI) at Mississippi State University. He holds a B.S. in Computer Engineering from the University of Alabama in Huntsville, an M.S. in Computer Science with a minor in Information Assurance and a Ph.D. in Computer Science at Auburn University. His research interests include: open-source security and reliability, test case prioritization and minimization, software fault modeling, and machine learning.

Box 9627, Mississippi State, MS 39762
Phone: (662) 325-2080
E-mail: pape@dasi.msstate.edu

Drew Hamilton is the Associate Vice President for Research at Mississippi State University and a professor of computer science and engineering. Previously he held faculty appointments at Auburn University and the US Military Academy and a visiting appointment at the US Naval Postgraduate School. Dr. Hamilton earned his doctorate in computer science from Texas A&M University. Dr. Hamilton is a distinguished graduate of the Naval War College.

Box 6343, Mississippi State, MS 39762
Phone: (662) 325-3570
E-mail: hamilton@research.msstate.edu

REFERENCES