

A Hybrid Approach to Propagation Analysis

David Byers Mariam Kamkar
Department of Computer Science
Linköping University
{davby,marka}@ida.liu.se

Abstract

Propagation analysis is a dynamic code analysis technique that can be used to quantitatively assess certain software properties that are otherwise difficult to assess, such as testability, safety and security. The currently accepted analysis technique relies heavily on repeated execution of the code being assessed, and is therefore very expensive to apply. We are currently developing static analysis methods to assist in the dynamic assessment. Our aim is to replace the most expensive parts of the dynamic analysis technique with less expensive and nearly as accurate static analysis techniques. This paper summarizes our ideas on how this can be done.

1 Introduction

When thinking about testing, the kind of testing that most often comes to mind is running software to determine if it functions correctly or not. Testing the functionality of a system is of course important, and is probably the most common type of testing. But there are other aspects of a system that need testing as well. System safety and security are two such aspects which are becoming increasingly important as software permeates through all aspects of life. Today we rely on software to fly us across oceans to Computer Science conferences, and we rely on software to keep our bank records intact. We rely on software to control our car on the way to work and we rely on it to keep confidential information safe from prying eyes.

Let us for a moment jump to another topic, and come back to the issues of testing for, or more correctly stated, assessing for safety and security. Testers know that the effort required to test a piece of software varies, and that the portion of faults actually found also varies. This property of software is called *testability*. Testability varies between software types. For example, fault tolerant software is notoriously difficult to test, since per definition, faults are not supposed to cause failures. But testability can also vary on a smaller level. Different choices in implementation can yield products that are identical in functionality and behavior, but have different testability. At this level there are many different ways of defining testability [1, 3, 6, 7]. The definition we like working with is the predicted minimum fault size [2, 7].

The size of a fault is simply the proportion of inputs that cause the fault to result in failure. The minimum fault size is the smallest such size in a program.

The predicted minimum fault size is an educated guess as to how large the actual minimum fault size is. When estimating the minimum fault size it is useful to consider how a fault causes failure. For a fault to cause a failure the fault has to be executed. The fault also needs to cause the internal state of the program to be different from what it should be; we call this infection. Finally the change in the internal state has to propagate to the program's outputs; this we call propagation. The probability of a failure due to a fault at a certain location is stated in equation 1. The size of the fault is simply this probability times the size of the input domain.

$$\Pr[\text{Failure}] = \Pr[\text{Execution}] \times \Pr[\text{Infection}|\text{Execution}] \times \Pr[\text{Propagation}|\text{Infection}] \quad (1)$$

It is obvious from this that the minimum fault size depends on the probability distribution of the program's inputs, the operational profile. When the operational profile changes, so do the fault sizes, and this also testability.

To estimate fault sizes, the software is repeatedly executed with random inputs. First the probability of execution is estimated by counting how many times each position in the program is executed. Next infection is estimated by randomly mutating each position in turn and recording the number of times the mutation caused the internal state to change. Finally the internal state is perturbed at each position in turn, and if the resulting output is different from the output of the same execution without the perturbation, propagation has occurred.

Testability assessments can be used in a number of different ways. Firstly they point out to testers where faults are likely to be missed. For these segments of code some other verification method than testing may be appropriate. Secondly a testability assessment can help determine how testing resources should be allocated to various parts of a system. A completely different, but very interesting application of testability is to enhance reliability estimates. If N random test cases yield no faults, the reliability of the system, r can be taken as $1/N$. If at the same time the testability is brought up to r , the probability that the software contains faults becomes 0 with a high confidence. Using this method, reliability estimates of as low as 10^{-7} could be made [1]. It should be noted that practical experience with this method has not been reported to any great extent, but the potential benefits make it an interesting alley to explore.

Moving back to assessing safety and security, we find that the same concepts used in testability, in particular propagation analysis, useful [4, 5]. When assessing safety the internal state of the program is perturbed, and the system is examined to determine if this perturbation results in an unsafe state. If that happens, there may be a safety problem in the system. Similarly, to assess security, the internal state is perturbed in a way that simulates an attack, and the operation of the system is monitored to ensure that the perturbation does not result in a breach of security.

These analyses all have the advantage of being completely automatic, but they are also expensive. In particular, propagation analysis is computationally expensive, and this limits its usefulness on larger systems. We have examined the possibility to perform at least part of the analysis using static methods rather than dynamic ones.

2 Static Execution Analysis

Estimating the probability of execution is the least expensive of the three analysis methods (execution, infection and propagation probabilities.) Currently it is used in testability analysis and will probably be required for hybrid propagation analysis. The dynamic execution probability estimation suffers from sometimes not executing a position in the code at all. When this happens, infection and propagation analysis cannot be performed. Instead the estimate becomes an upper bound derived from the number of times the software has been executed. In order to continue the analysis the portion of code that has not been executed needs to be manually verified. Once it is known to be correct it can be excluded from the analysis, which will lead to a higher testability.

We hoped to develop a static analysis method for calculating the execution probability. To be viable the method would have to be nearly as inexpensive to use as dynamic analysis. The method we developed turned out not to be accurate enough and to be more expensive than dynamic analysis.

3 Static and Hybrid Propagation Analysis

Propagation analysis is the most expensive of the three analyses, and also the most useful. An approach to static propagation analysis has been proposed but not thoroughly examined [1]. Essentially the idea is to perform forward data-flow analysis, marking variables that are dependent on the infected variable. This way a fault set is built for each program position. The proposed hypothesis is that when the size of the fault set is large enough, propagation is virtually guaranteed. By differentiating between variables that are certain to be in the fault set and those that may be in the fault set, and calculating the constraints on the inputs that are required to reach a location that is certain to affect the program's output, and calculating the probability that the constraints are met, it is possible to improve the fault set analysis. The constraint and probability calculations involved can be simple in some cases, but very expensive in other cases.

Our approach is to combine inexpensive dynamic analysis with inexpensive static analysis to achieve the same results. Dynamic estimation of execution probability can replace the constraint calculation, and to avoid the assumption that fault sets always increase, we propose to use soft fault sets, where variables are included with a given confidence, and dynamic local propagation analysis.

Soft fault sets are simply fault sets where each variable has an associated probability. The probability denotes the likelihood that the variable should really be included in the fault set. The initial fault set, which contains the perturbed variable is $(x_{1.0})$. After calculation on x , additional variables may be included with equal or lower probability. For example, if the calculation were $z = x \bmod 10$, and x was perturbed, the fault set at that location would be $(x_{1.0} z_{0.9})$ since a random change in x would not change z 10% of the time.

In order to propagate soft fault sets we need a propagation estimate for each program location and variable, what we call a local propagation estimate. We propose to use dynamic analysis to calculate such an estimate. For each location, its input variables are perturbed in turn, and we check if this perturbation causes any other change in the internal state of the program. For example, the function

in figure 1 contains two code locations. At location one, simulation would find that when either x or y is perturbed, propagation almost always occurs. At location 2, when d is perturbed, simulation will find that propagation occurs only 50% of the time.

Using the example in figure 1 again, this time estimate the probability of propagation, we start by perturbing x at location 1. Thus, entering location 1, the fault set is $(x_{1.0})$. Since local propagation tells us that a perturbation in x will propagate to d , the fault set before location 2 is $(x_{1.0}, d_{1.0})$. The local propagation data for location 2 tells us that a perturbation of d is only propagated 50% of the time, yielding the fault set $(x_{1.0}, d_{1.0}, f()_{0.5})$ which results in a propagation estimate of 0.5 for location 1. In contrast, using simple fault sets and no local propagation data, the result would have been a propagation estimate of 1.0 for location 1.

```

double f(int x, int y)
{
    int d;

(1)      d = round(sqrt(x * x + y * y)); (x:1, y:1)
(2)      return d % 2;                  (d:0.5)
}

```

Figure 1: Example of local propagation analysis

4 Future Work

This paper reports on ideas we have for improving the techniques used for assessing software testability, and for testing software safety and security. In the near future we will be exploring our idea of soft fault sets and local propagation analysis. Our focus will be on making the results of the analysis correlate well with dynamic analysis and to increase the efficiency of the local propagation analysis to the point where our hybrid method is less expensive to apply than the purely dynamic analysis. We will also try to find heuristics for speeding up or eliminating local propagation analysis where possible. For example, if we could identify code locations with a 100% probability of propagation, those locations could be excluded from dynamic analysis with no loss of precision.

Further on we will reexamine static estimation of execution probability and attempt to define and implement a static analysis method for estimating the probability infection.

References

- [1] Roy S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, June 1991.

- [2] Michael A. Friedman and Jeffrey M. Voas. *Software Assessment: Reliability, Safety, Testability*. John Wiley & Sons, Inc., 1995.
- [3] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.
- [4] Jeffrey M. Voas. Testing software for characteristics other than correctness: Safety, failure tolerance and security. In *Proceedings of the 13th International Conference on Testing Computer Software*, June 1996.
- [5] Jeffrey M. Voas, Anup K Ghosh, Gary McGraw, and F. Charron. Defining an adaptive software security metric from a dynamic software failure tolerance measure. In *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS'96)*. IEEE Computer Society, 1996.
- [6] Jeffrey M. Voas and Keith W. Miller. Semantic metrics for software testability. *Journal of Systems and Software*, 20(3):207–216, March 1993.
- [7] Jeffrey M. Voas and Keith W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, May 1995.