Model driven testing distributed environment monitoring system

Yan Li[1], Zhe Zhang[2*], Guihong Jiang[1], Xiaofeng Cui[1]

[1]*School of Computer, Shandong University of Technology, Zibo 255049, Shandong Province, China*

[2]*Software College, Nanyang Normal University, Nanyang 473000, Henan Province, China*

Received 6 October 2013, www.tsi.lv

Abstract

Distributed environment monitoring system is more and more widely used, especially the design and verification of embedded system in environmental monitoring is the guarantee of successful use of environmental monitoring. In this paper we demonstrate how test-case prioritization can be performed with the use of model-checkers. For this, different well known prioritization techniques are adapted for model-based use. New property based prioritization techniques are introduced. In addition it is shown that prioritization can be done at test-case generation time, thus removing the need for test-suite post-processing. Several experiments for embedded systems are used to show the validity of these ideas.

Keywords: test case prioritization, software testing, model checking, property testing

1 Introduction

In today's society, environment monitoring system is being paid more attention. Distributed environment monitoring system is more and more widely used [1], especially the design and verification of embedded system in environmental monitoring is the guarantee of successful use of environmental monitoring.

It has been shown [2, 3] that the order in which the test-cases of a test-suite are executed has an influence on the rate at which faults can be detected. In this paper we demonstrate how test-case prioritization can be performed with the use of model-checkers. As common prioritization techniques are based on program source-code, these techniques have to be to adapted to the model-based setting. In addition, new property based prioritization techniques made possible by the use of model-checkers are introduced.

Obviously, a model-checker based method to test case prioritization is a useful addition to model-checker based test-case approaches. We therefore show how prioritization can be done at test-case generation time when using model-checkers to create test-cases. That way, no post-processing of the test-suites is necessary while still achieving an improved fault detection ratio of the resulting test-suite. The ideas described in this paper are illustrated using several example applications.

This paper is organized as follows: Section 2 recalls the principles of test-case prioritization and presents different prioritization techniques for model-based use. Then, Section 3 describes how prioritization is performed with the help of a model-checker, while Section 4 shows how prioritization can be done at test-case creation time. Section 5 describes our experiment for several security-critical embedded systems and presents the results achieved. Finally, Section 6 concludes the paper.

2 Test case prioritization

Test-case prioritization is the task of finding an ordering of the test-cases of a given test-suite such that a given goal is reached faster. The test-case prioritization problem is defined by Rothermel et al. [4] as follows:

Given: T, a test-suite; PT, the set of permutations of T; f a function from PT to the real numbers.

Problem: Find $T' \in PT$ such that

$$(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') > f(T'')] \quad (1)$$

PT is the set of all possible orderings of T, and f is a function that yields an award value for any given ordering it is applied to. f represents the goal of the prioritization. For example, the goal might be to reach a certain coverage criterion as fast as possible, or to improve the rate at which faults are detected. There are different test-case prioritization techniques that can be used to achieve such goals.

Several different prioritization methods have been discussed in previous works [5, 8]. These methods are generally based on the source code of a program, e.g., the coverage of statements or functions. In contrast, when using a model-checker to determine prioritization we base the techniques on a functional model of the program to test. This section does not provide a complete overview of all available prioritization techniques but selects a representative subset that can be used to illustrate the usefulness of model-checkers in the prioritization process. In addition, the use of a model-checker allows new kinds

* ***Corresponding author*** e-mail: jdd35@163.com

of prioritization techniques which are introduced in this section.

2.1 TOTAL COVERAGE PRIORITIZATION

There are several code-based prioritization methods that sort test-cases by the number of statements or functions they cover. Model-checker based testing allows the formulation of coverage criteria as properties, as described in the next section. We therefore generalize from different code based methods to a coverage based method which is applicable to any coverage criterion expressible as a set of properties.

For example, the model-based coverage criterion Transition Coverage requires that each transition in an automaton model is executed at least once. Test-case prioritization according to transition coverage sorts test-cases by the number of different transitions executed.

2.2 ADDITIONAL COVERAGE PRIORITIZATION

Total Coverage Prioritization achieves that those test-cases with the biggest coverage are executed first. This does not necessarily guarantee that the coverage criterion is achieved as fast as possible. Additional coverage prioritization first picks the test case with the greatest coverage, and then successively adds those test-cases that cover the most yet uncovered parts.

2.3 TOTAL FEP PRIORITIZATION

This technique orders test cases by the ability to expose faults (fault exposing potential). Mutation analysis [6] is used to determine these values. For a given program a set of mutants is created by the application of a set of mutation operators. Each application of a mutation operator creates a mutant of the source code that differs from the original by a single valid syntactic change. The mutation score represents the ratio of mutants that a test-suite can distinguish from the original program. This mutation score can be calculated for each test-case separately, and then used as an award value for test-case prioritization. Total FEP prioritization uses the mutation score for a total sorting

2.4 ADDITIONAL FEP PRIORITIZATION

Similarly to additional coverage based prioritization test-cases can be sorted by the number of additional, yet undetected mutants. First the test-case with the highest mutation score is chosen, and then successively those test-cases are added that maximize the total number of detected mutants. Traditionally, this FEP based prioritization is computationally more complex than coverage based methods.

2.5 TOTAL PROPERTY PRIORITIZATION

This is a new technique made possible by the use of model-checkers. It is based on the idea of property relevance [7]. A test-case consists of values that are used as input data for the system under test, i.e., they represent the inputs the system receives from its environment. A test-case is said to be relevant to a requirement property if a property violation is possible when the input values are provided to an erroneous implementation. In practice, this can be determined by checking whether there is a mutant that can violate the property. A test-case can of course be relevant to more than one property. Total property prioritization sorts test-cases by the number of properties they are relevant to.

2.6 ADDITIONAL PROPERTY PRIORITIZATION

Similarly to the previous techniques, this method begins with the test-case that is relevant to the most properties and then successively adds test-cases that are relevant to yet uncovered properties.

2.7 HYBRID PROPERTY PRIORITIZATION

If the number of properties is significantly smaller than the number of test-cases, then a property based prioritization can quickly achieve property coverage. In general, the prioritization of the remaining test-cases starts again with the test-case with the highest award value. However, it is also conceivable to combine two different award functions. For example, it can be useful to sort test-cases totally based on the number of relevant properties, and then use a coverage prioritization as a secondary sorting method within test-cases of equal property relevance. We use transition coverage as secondary award value in our experiments.

2.8 RANDOM PRIORITIZATION

Random prioritization is interesting for evaluation of the different techniques. In average, any sorting method should achieve better results than random prioritization in order to be useful. We therefore use random prioritization as a lower bound for our analysis.

2.9 OPTIMAL PRIORITIZATION

The optimal prioritization sorts test-cases such that a given set of faults is detected with the minimum number of test-cases. This technique is not applicable in practice as it requires a-priori knowledge about the faults that are to be exposed. However, in experiments with known mutants it serves as upper bound for improvements that can be achieved with prioritization.

3 Using model checking to determine prioritization

In this section we show how the prioritization methods presented in the previous section can be performed in practice. As mentioned, we use model-checkers for prioritization. In order to do so it is necessary to re-formulate test-cases as models, which allows analysis with regard to certain properties. This can be easily done by basing the transition relation of all variables on a special state-counting variable, as suggested by Ammann and Black [1]. As an example, assume a simple test-case

$t = \{(x = 1, y = 0), (x = 0, y = 1), (x = 1, y = 1)\}$.

Using the input language of the model-checker NuSMV [5] which we used for our experiments, the test-case can be expressed as:

```
MODULE main
VAR
  x: boolean;
  y: boolean;
  State: 0..2;

ASSIGN
  init(x):=1;
  next(x):= case
    State = 0: 0;
    State = 1: 1;
    1: x;
  esac;

init(y):=0;
next(y):= case
  State = 0: 1;
  State = 1: 1;
  1: y;
esac;

init(State) := 0;
next(State) := case
  State<2: State+1;
  1: State;
esac;
```

3.1 COVERAGE PRIORITIZATION

Model-based coverage criteria can be expressed as trap properties [9, 10]. For each coverable item one such property is formulated, expressing that the item cannot be reached. For example, a trap property might claim that a certain state is never reached or that a certain transition is never taken. Challenging a model-checker with a model and a trap property results in a counter-example, which is a trace illustrating how the item described by the trap property is reached. This principle is used for test-case creation, where it automatically results in test-suites that achieve a given coverage criterion. It is also used to measure the coverage of test-suites. The test-cases are converted to models as described above, and then the model-checker is challenged with the resulting models and the trap properties. For each trap property that results in a counter-example it is known that the test-case covers the according item.

While for overall coverage measurement it is sufficient to check how many trap properties are violated, this can easily be extended such that each test-case is checked against all trap properties. That way the overall coverage of each test-case can be determined. This information can be used in order to sort test-cases according to their coverage, either totally or additionally. The prioritization works as follows:

1) Create models from test-cases.
2) Create trap properties ***TP*** from coverage criterion.
3) **for** each test-case model *t* **do.**
4) model-check *t* against ***TP.***
5) each trap resulting in a counter-example is covered.
6) **end for**
7) sort test-cases by number of covered traps.

3.2 FEP PRIORITIZATION

Fault exposing prioritization is based on mutation analysis. Model and specification mutation was introduced by Ammann and Black [11]. The ability to expose faults can be measured as the mutation score of a test-case.

With model-checkers, this can be done in two ways. One option is to create mutants of a given model, and then symbolically execute the test-cases against these models by combining the mutant model and the test-case model, using the test-case values as input-values for the mutant. A mutant is detected if the model-checker returns a counterexample when queried whether the output values of mutant and test-case are equal along the test-case. Unlike coverage based methods, this requires the model-checker to use the actuOrial model in addition to the test-case model. If the model is complex, then this process is less efficient than the coverage based method.

The alternative is to reflect the transition relation of the model as special properties [12]. Each reflected property refers to one variable. For each possible transition a variable can take, there is one such property. It consists of the transition condition and makes an assertion about the value of the variable in the next state. These reflected properties can then be mutated instead of the original model. When checked against the original model the mutated properties result in efficient test-suites [2]. A mutation score can be efficiently calculated by checking these properties against the test-case models. This prioritization is therefore identical to coverage based prioritization apart from the use of mutated reflected properties instead of trap properties.

3.3 PROPERTY PRIORITIZATION

Property prioritization uses the concept of property relevance. A test-case is relevant to a property if the execution of the test-case can theoretically lead to a violation of the property. As presented in [13], property relevance can be determined with the aid of a model-checker by symbolically executing the test-case against a modified model which is allowed to take one single erroneous transition. The model checker then efficiently determines if a single erroneous transition is sufficient in order to reach a property violating state during the test-case execution. This process has to be repeated for each test-case.

1) Create modified model M' from model M .
2) Create models from test-cases.
3) **for** each test-case model *t* **do.**

4) Combine t and M' such that M' takes input values from t instead of the environment.
5) Model-check M' against all requirement properties.
6) t is relevant to each property causing a counter-example.
7) **end for**
8) Sort each test-case by relevance.

While the complexity of this evaluation process can be higher than for coverage or reflection based methods, it is only necessary to challenge the model-checker once with each test-case, so this is still significantly more efficient than the determination of the mutation score using symbolic execution would be. Once the property relevance of each test-case has been determined, this information can be used in order to calculate a total or adding prioritization for the test-cases.

3.4 OPTIMAL PRIORITIZATION

The optimal execution order of a test-suite with regard to a set of mutants is calculated with a greedy algorithm that successively adds the test-case next that detects the most yet undetected mutants.

4 Prioritizing test case at creation

Each test-case is assigned an importance value, initially 1. If a test-case is a prefix of another test-case or equal to it, the importance of this other test-case is increased. If a test-case subsumes other test-cases, then its importance is the sum of the subsumed test-cases plus 1.

1) **while** t =ceate next test-case **do**
2) importance of t =1
3) **if** $\exists t' \in T : t = t'$ **then**
4) increase importance of t' by 1
5) **else if** $\exists t' \in T : t \subset t'$ **then**
6) increase importance of t' by 1
7) **else if** $\exists t' \in T : t \supset t'$ **then**
8) **for** all $\exists t' \in T : t \supset t'$ **do**
9) replace t' with t in T
10) increase importance of t with importance of t'
11) **end for**
12) **else**
13) inset t in T
14) **end if**
15) **end while**
16) sort test-cases by importance

When creating test-cases automatically it is often the case that redundant test-cases are created. If a new test-case is a prefix of another test-case it is sufficient to retain the subsuming, longer test-case. If a new test-case subsumes other test-cases it is sufficient to retain the new test-case. Redundant test-cases are usually discarded. However, this redundancy information can also be used to prioritize test-cases. If a test-case or part of it is created more than once, this can be seen as an indication that this test-case is more important than other test-cases. With this information prioritization can be performed without post-processing of the test-suite.

5 Experimental results for three crucial embedded systems

This section presents the results of an empirical evaluation for three security-critical embedded systems aiming to show that model-based test-case prioritization results in a noticeable performance improvement. We also want to analyze the newly defined property coverage techniques in comparison to well-known techniques. Finally we want to determine whether prioritization at test-case generation time results in a measurable improvement.

5.1 APFD

In order to quantify the efficiency gains achieved with a certain test-case prioritization, the metric APFD was introduced by Rothermel et al. [13, 14]. This metric is the weighted average percentage of faults detected over the life of a test suite. The APFD of a test suite T consisting n test cases and m mutants is defined as:

$$APFD = 1 - \frac{TF_1 + TF_2 + TF_3 + \cdots + TF_m}{nm} + \frac{1}{nm}. \qquad (2)$$

Here, TF_i is the first test-case in ordering T_0 of T which reveals fault i. We use this metric in order to compare the different prioritization techniques.

5.2 EXPERIMENT SETUP

The evaluation is based on a set of three examples. Each example consists of an SMV-model and specification. Different model-checker based methods (various coverage criteria, different mutation operators, property based methods) are used in order to create 23 different test-suites for each model. For each model a set of mutants is created. Unlike for program mutation, a model-checker can efficiently determine whether a model mutant is equivalent to the original or not. The APFD values for each of the test-suites is calculated using the subset of the in-equivalent model mutants that can be detected by the test-suite. Table 1 sums up the results of the test-case generation and the numbers of detected mutants. Only detectable mutants are relevant for the determination of the APFD value, as the test-case execution order has no influence on undetectable mutants.

Car Control (CA) is a simplified model of a car control. The Safety Injection System (SIS) example was introduced in [3] and has since been used frequently for studying automated test-case generation. Environmental Control (CC) is based on [12]. In order to validate the method we also use a set of 25 erroneous mutant implementations for the Cruise Control example applications written by Jeff Offutt.

TABLE 1 Test-suite statistics

Example	CA		SIS		EC	
	Avg	Max	Avg	Max	Avg	Max
Test Cases	51	243	22	85	35	246
Mutants	264	311	265	339	535	732

5.3 RESULTS

Following the tradition of previous papers about test-case prioritization we use box-plots to illustrate the results of the APFD analysis. The box-plots illustrate minimum, maximum, median and standard deviation for each of the used prioritization methods. As can be seen in Figures 1, 3 and 4, there is still a gap between all prioritization techniques and the optimal prioritization.

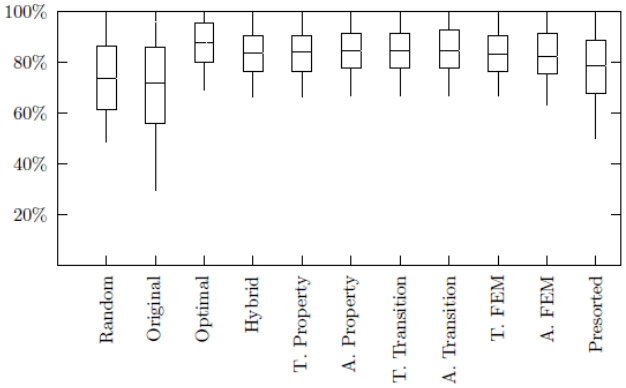

FIGURE 1 APFD of cruise control model

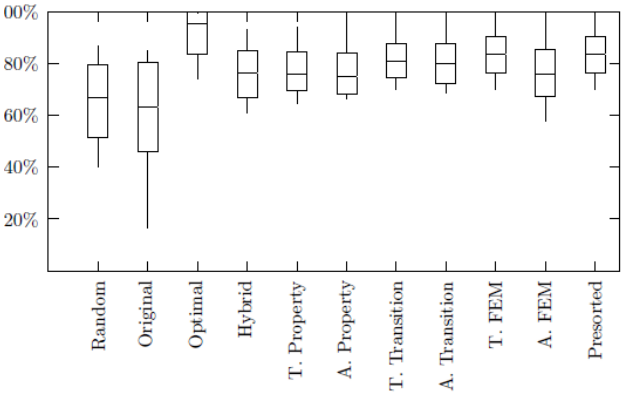

FIGURE 2 APFD of cruise control implementation

However, there is an improvement clearly visible compared to the random sorting of the test-cases and the original sorting, as provided by the test-case generation algorithm. Figure 6 lists the average APFD values for all examples and methods in a concise manner. The improvement is not always as significant as reported in previous works. This is probably because we used test-suites of different sizes, and the improvement is not quite so obvious for large test-suites. In general, a large amount of the mutants is detected with the first couple of test-cases (Figure 5), yet the remaining test-cases and mutants can distort the APFD value, if there are many test cases. Nevertheless, an improvement is visible. Figure 2 illustrates the APFD values for the same test-suites (except the optimal one) as in Figure 1, executed with the 25 erroneous implementations of Cruise Control. The values are comparable and we conclude that model-based prioritization is also valid for real implementations.

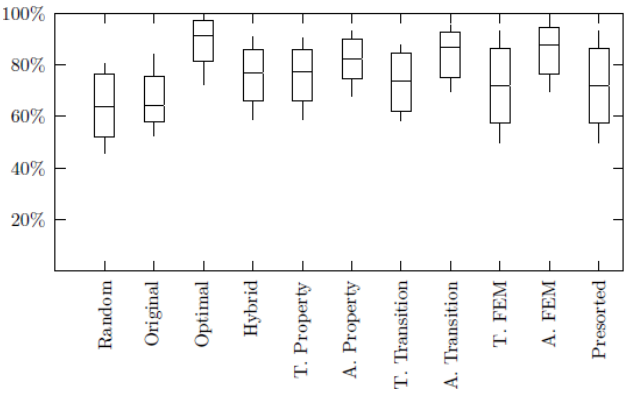

FIGURE 3 APFD of environmental system

The prioritization performed at test-case generation time (labelled presorted in the figures) is clearly better than random ordering, however there is still a gap between presorted test-suites and post-processing prioritization. This gap is also visible in Figure 5. Interestingly, the presorted test-suites performed better than most other prioritization techniques during the evaluation on the cruise control implementations. In general we can conclude that prioritization at test-case generation is definitely useful, especially as it only requires negligible additional computational costs.

There are only minor differences between the various prioritization techniques. In general, those techniques that use adding sorting perform slightly better than those with total sorting. Property prioritization performs good (regard also Figure 5), in fact it sometimes outperforms coverage based prioritization techniques.

However, this case study does not reflect on the quality of the specification. It is conceivable that a specification consisting of more and better properties will result in better property based prioritization.

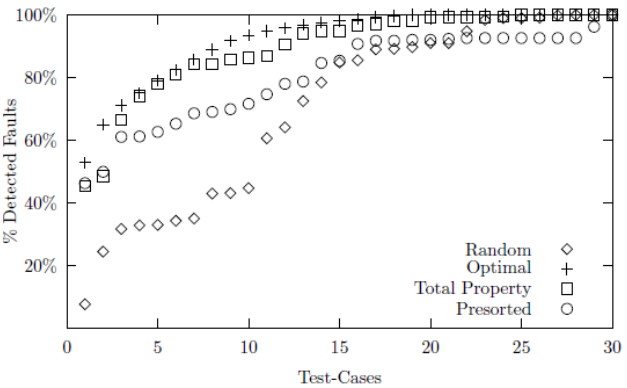

FIGURE 4 Fault detection rates for environmental control example

While model-checkers in general are prone to performance problems this is not a problem for prioritization, as the state space of test-case models is usually significantly smaller than that of related functional models.

6 Conclusion

In this paper we have demonstrated how model-checkers can be used for test-case prioritization for the embedded systems of environmental control.

This makes it possible to efficiently apply prioritization when creating test-cases with model-checkers. We adapted several well known prioritization methods originally based on source code to models. In addition we introduced new property based prioritization methods. Finally, we showed that test-case prioritization can be performed automatically during test-case generation, without post-processing.

References

[1] Hu C, Zhu L 2010 The analysis and the evaluation of complicated network software *LNCS* **13**(10) 1-5

[2] Wang Y, Xia H, Yan R 2008 The analysis of the social network and the study of the application cases of NetDraw *Modern education technology* **18**(4) 85-9

[3] Pothen A, Simon H, Liou K P 1990 Petitioning sparse matrices with eigenvectors of graphs. S*IAM Matrix Anal. Appl* **11** 430-6

[4] Girvan M, Newman ME 2001 Community structure in social and biological networks. *Proc. Natl. Acad. Sci* **99**(12) 7821-6

[5] Newman ME, Girvan M 2004 Finding and evaluating community structure in networks *Phys. Rev. E* **39**(10) 69-84

[6] Toyoda M, Kitsuregawa M 2003 Extracting evolution of web communities from a series of web archives *Proceedings of the fourteenth ACM conference on Hypenext and hypermedi*a 78-87

[7] Palla G, Derenyi I, Vicsek T 2007 The Critical Point of *k-groups* Percolation in the Erdos–Renyi Graph *Journal of Statistical Physics* **128**(1) 219-27

[8] Palla G, Barabasi A-L, Vicsek T 2007 Community dynamics in social networks *Noise and Stochastics in Complex Systems and Finance* **6601**(3) 273–87

[9] Xu C, Zhang Y, Dan Y 2011 Ontology based Image Semantics Recognition using Description Logics *IJACT International Journal of Advancements in Computing Technology* **3**(10) 1-8

[10] Ju C, Wei J 2012 Research on Multi-interest Profile Based on Resource Clustering *JCIT Journal of Convergence Information Technology* **7**(21) 582-90

[11] Gargantini A, Heitmeyer C 1999 Using Model Checking to Generate Tests From Requirements Specifications *In Software Engineering - ESEC/FSE'99: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering* **1687**(1) 146-62

[12] Kim J-M, Porter A 2009 A history-based test prioritization technique for regression testing in resource constrained environments. *In ICSE '09: Proceedings of the 31th International Conference on Software Engineering* **139**(3) 119-29

[13] Rothermel G, Untch R H, Chu C, Harrold M J 2009 Test case prioritization: an empirical study *Proceedings of the IEEE International Conference on Software Maintenance* **168**(1) 179-83

[14] Srikanth H, Williams L 2005 On the economics of requirements-based test case prioritization *Proceedings of the 7th international workshop on Economics-driven software engineering research* **153**(1) 1-3

Authors

Yan Li, born in April, 1977, Zibo County, Shandong Province, China

Current position, grades: the lecturer of School of computer, Shandong University of Technology, China.
University studies: B.Sc. in application of computer Technology from Northeast Forestry University, M.Sc. from Shandong University in China.
Scientific interest: computer modelling, information retrieval.
Publications: more than 10 papers.
Experience: teaching experience of 14 years, 3 scientific research projects.

Zhe Zhang, born in January, 1982, Nanyang County, Henan Province, China

Current position, grades: the lecturer of School of Software, Nanyang Normal University, China.
University studies: M.Sc. in computer applications from Huazhong University of Science & Technology in China.
Scientific interest: software engineering, formal modelling.
Publications: more than 6 papers.
Experience: teaching experience of 10 years, 4 research projects.

Guihong Jiang, born in November, 1966, Zibo County, Shandong Province, China

Current position, grades: the associate professor of School of computer, Shandong University of Technology, China.
University studies: B.Sc. in Organic chemical Engineering from Qingdao University of Science & Technology in China.
Scientific interest: database design, software engineering.
Publications: more than 6 papers, 8 teaching books about database or program design published.
Experience: teaching experience of 25 years, 2 scientific research projects.

Xiaofeng Cui, born in October, 1969, Zibo County, Shandong Province, China

Current position, grades: the associate professor of School of computer, Shandong University of Technology, China.
University studies: B.Sc. in computer science from China Agricultural University.
Scientific interest: data mining, artificial intelligence.
Publications: more than 4 papers.
Experience: teaching experience of 17 years, 10 education research projects.