5-2013

SOFTWARE FAULT DETECTION VIA GRAMMAR-BASED TEST CASE GENERATION

Songqing Liu
University of Nebraska at Omaha

SOFTWARE FAULT DETECTION VIA GRAMMAR-BASED TEST CASE GENERATION

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfilment
of the Requirements for the Degree

Master of Science in Computer Science

University of Nebraska at Omaha

by

Songqing Liu

Omaha, Nebraska

May, 2013

Supervisory Committee:

Haifeng Guo, Ph.D.

Harvey Siy, Ph.D.

Haorong Li, Ph.D.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

SOFTWARE FAULT DETECTION VIA GRAMMAR-BASED TEST CASE GENERATION

Songqing Liu, M. S.

University of Nebraska, 2013

Advisor: Haifeng Guo, Ph.D.

Fault detection is helpful to cut down the failure causes by logically locating and eliminating defects. In this thesis, we present a novel fault detection technique via structured input data which can be represented by a grammar. We take a set of well-distributed test cases as input, each of which has a set of test requirements. We illustrate that test requirements come from structured data can be effectively used as coverage criteria to reduce the test suites. We then propose an automatic fault detection approach to locate software bugs which are shown in failed test cases. This method can be applied in testing data-input-critical software such as compilers, translators, reactive systems etc. Preliminary experimental study proves that our fault detection approach is able to precisely locate the faults of software under test from failed test cases.

DEDICATION

To my parents, my family members, mentors and friends who have accompanied and inspired me on my journey

ACKNOWLEDGMENTS

I would like to thank the many people who kindly assisted me throughout my master program. Special thanks to my major advisor, Dr. Haifeng Guo, for his help, guidance, patience and support throughout my graduate study. I will be forever thankful for and appreciative of his encouragement. I would also like to thank my master committee members who have gone through the entire process with me: Dr. Harvey Siy, Dr. Haorong Li, for their contribution to the development of my graduate education. The professors and staff of the Computer Sciences Department have been especially kind and helpful during my time at UNO. Many thanks go to my colleagues working in IST at PKI: Mr. Weng Zheng, Mr. Bo Guo, Ms. Yushu Song and Mr. Aruna Weerakoon, for their hand-by-hand help, support and friendship. I am also grateful to Dr. Zongyan Qiu for all the technique supports and many valuable conversations. Without their efforts, much of this work would not have been possible. A special thanks goes to Dr. Harvey for his generous instructions to me on both scientific idea and technology, even from the very first step, and with great patience.

Finally, this dissertation would not be possible without the support of my family. I dedicate this work to my wife, Sissy Zhang, my daughter Summer Liu and my parents. Words cannot explain how much I love and respect my whole family for everything they have done for me.

I could not have done this work without the support and help of all these people. Thank you very much.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems continue to grow in size and complexity as more functionality is developed and more integration is needed. At the meantime, software bugs could exist anywhere of software. Not only do they infect software at development phase, but they also breakout software testing attempts and hide bugs into product code. In this sense, fault detection is a critical task in software testing throughout the whole life cycle of software development.

1.1 Background Research

Traditional software fault detection is a formidable task that costs time, effort, and a comprehensive knowledge of the source code. For a complicated system, developing methods for detecting the fault of software is extremely critical and difficult. As the efficient fault removal technique, fault detection is one of the most effort-intensive activities during software development [7]. In recent years, the field of automated fault detection has made significant progress.

One such fault detection technique is static analysis, the process of evaluating a

system or component based on its form, structure, content, or documentation [31], which does not require program execution. Lots of these techniques are proposed [22],[25],[26],[28],[27],[5],[15],[9] to automate the identification of anomalies that can be revealed via static analysis, such as uncaught runtime exceptions, redundant code, inappropriate use of variables, division by zero, and potential memory leaks.

Another important branch of research in automated fault detection is formed by experimental approaches. These techniques systematically alter applied changes [37], input [39], or object interaction [8] in order to narrow down failure causes to a small fraction of the search space. Most existing methods for automatically fault localization rely on analysis of execution traces. While such experimental techniques can precisely pinpoint failure causes, they can also alter program behavior in a way that is impossible to achieve in the original setting. Adopting delta debugging on program states [39], for example, freely produces impractical states, and depends on the run-time system to locate inconsistencies

Also, GD Fatta et al. [12] present a method to enhance fault localization for software system based on a frequent pattern mining algorithm. The test executions are recorded as function call trees. Based on test oracles the tests can be classified into successful and failing tests. A frequent pattern mining algorithm is used to identify frequent subtrees in successful and failing test execution. And D Jeffrey et al. [21] use additional coverage information of test cases to selectively keep some additional test cases in the reduced suites that are redundant with respect to the testing criteria used for suite minimization, with the goal of improving the fault detection effectiveness retention of the reduced suites.

Automatic test generation can significantly reduce the cost of software development and maintenance. In our approach, we use *Gena*, an automatic grammar-based test generator, which takes inputs a symbolic grammar and a total number of test

case to request, to produce well-distributed test cases for software testing. A symbolic terminal, highlighted by a pair of square brackets, is an abstract notation for a finite domain. A simple example of a symbolic grammar is shown in Figure 1.1 , where [N] is a symbolic terminal, whose instance can be any number between 1 and 1000. An example of test generation based on leftmost derivation would be as shown in 1.2

$$\begin{aligned} E &::= [N] \mid E + E \mid E - [N] \\ [N] &::= 1 \,..\, 1000 \end{aligned}$$

Figure 1.1: Symbolic Context-free Grammar 1

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow E - [N] + E \\ &\Rightarrow [N] - [N] + E \\ &\Rightarrow 359 - [N] + E \\ &\Rightarrow 359 - 54 + E \\ &\Rightarrow 359 - 54 + [N] \\ &\Rightarrow 359 - 54 + 823 \end{aligned}$$

Figure 1.2: Derivation Procedure of Simple Example

Test suite reduction uses test requirement to determine if the reduced set maintains the original suite's requirement coverage. Then we examine *granularity-based* customized test requirements for the fault detection problem, try to locate the fault of software as earlier as possible.

Our approach complements these existing techniques by systematically narrowing down the test input space to the simplest grammar instantiations that cause failure, thereby significantly reducing the traces to examine.

1.2 Problem Statement and Our Approach

Software Fault detection is a subfield of software testing which concerns identifying the bugs of software when they were triggered, and pointing out the type of fault and its precise location in software by executing test cases and observing the programs behavior. Given a program, fault detection is the problem of determining whether the program has bugs, preferably with test cases that trigger such bugs. Fault detection tools such as Valgrind [4], Find-Bugs [2], Fortify [3], Coverity [1], and many other tools are widely used in software development today.

Given a program and a test suite, an input for which program could encounter a failure, fault detection is the problem to identifying the reasons, including the location, cause, and possible fixes for the failure. Fault detection is an important and time-consuming step in debugging software failure. Currently, fault detection is mostly a manual process, either during development (statistical analysis), or during program-running time (experimental analysis). The latter one is especially hard to diagnose software because it is difficult to reproduce the failure at the developers site and because privacy and economic concerns severely limit what information is available from the en-users site.

In this thesis, we present a novel automated fault detection approach to identify software bugs, using grammar-based test case generation where each test case has an associate set of test requirements.

Our analysis follows the steps that we summarize below

- Use grammar-based test generation to generate inputs. Each generated test case has an associate set of test requirements
- Test suites reduction using greedy approach based on test requirements

- Fault detection of software

1.3 Contributions and Significance

We present a new approach on automated fault detection to localize the defects of software under test which is based on a grammar-based test generation where each generated test case has an associate set of test requirements. Our experimental results show that this method can be used in the software with structured data as input which these structured data can be easily presented by grammar.

We show that structured testing requirements, generated along with test cases, can be effectively used as coverage criteria for test suite reduction. Our fault detection approach is able to explore structural features in a systematic way to locate a set of *common least sub-structures*, each of which can cause testing failures.

1.4 Organization

The remaining part of this thesis is structured as follows: chapter 2 gives a brief introduction of the structured testing requirements from grammar-based test case generation. Chapter 3 describes the procedure of test suites reduction, illustrates how we can use test requirements as coverage criteria to reduce the test suites. Chapter 4 presents a novel approach of fault detection using structured test requirements to precisely detect the type of the fault, include location, cause and possible fix of the fault. Chapter 5 shows our preliminary experimental results on test reduction, fault detection and granularity analysis of software testing. Chapter 6 gives some more information on related works. Finally, the conclusions are given in chapter 7.

Chapter 2

Structured Testing Requirements

2.1 Test Coverage Criteria

Software test coverage criterion, widely researched as well as a large amount of software testing and test generation approaches, typically specifies testing requirements in terms of identified features of the software specification or the system under test. The common knowledge is that software test coverage criteria need use source information from either specification-based [40], which specifies the required testing in terms of identify features of the specification or the requirements of the software; or program-based, which takes testing requirements in terms of the program under test and decides if a test set is adequate according to whether the program has been thoroughly exercised.

Test requirement not only provides insight in the status of a requirement in a software development phase, but also is critical in software testing phase. For instance, the requirement coverage include whether or not a requirement is covered by an acceptance test, by a design artifact, by a system test etc. Joan C. Miller et al. [29] proposes an approach of code coverage analysis that using the logical tree, a

systematic way to test all possible combinations of input data, and even all portions of a given program. H. Kelly J. et al.[23] provide a practical approach to assessing modified condition/decision coverage (MC/DC) for aviation software products. Both of their code coverage methods check whether the software under test has been thoroughly reached at all code statements and all executive branches.

For data-flow coverage analysis, Fankl et al.[13] extend the definitions of the previously introduced family of data flow testing criteria and then define a family of adequacy criteria called feasible data flow testing criteria. M.J. Harrold et al. [17] incorporate the representative set algorithm into data flow testing system. Data flow analysis determines the relationship between definition of variables and uses of the same variables where we have the precondition that different associations of definition-use pairs would make different influence at software execution stage.

2.2 Test Requiremetns in Software Testing

Test requirements are very important in software testing which identify what object need to be tested and what goal are going to be validated by testers. Usually test requirements come from business requirements, functionality of the software, and internal relationship between different components etc. through market specifications, software function specifications and technical specifications of system.

An automatically generated test case often has a set of test requirements. However, the generated order of these two sets is not certain. Test requirements are often generated along with the generation of test cases, while sometimes, test requirements are generated before test case generation. In this circumstances, whenever the software is under testing while running these test cases, we will able to trace which features of the system have been tested.

Test requirements are also populate used in the areas of test case minimization, selection and prioritization [36]. Take test case prioritization as an example, P.R.Srivastva et al. [32] propose test prioritization technique prioritizes the requirements instead of prioritizing the test cases on the basis of requirements identified that can occur in a software project. Another view of the use of test requirements is that in automated model-based test case generation where test cases are purely produced according to a dataflow model, definition-use pairs of variables [16] are commonly identified as test case requirements using as efficient reduction criteria.

2.3 Grammar-Based Testcase Generation (GBTG) along with an Associate set of Test Requirements

GBTG is an approach to use symbolic context-free grammars to create a set of test cases. It is helpful on testing those applications which require structured data as inputs, such as compliers which take programs as input data, translator need files as input data, and web applications need sequences of events as data input. These structured data can be represented by a symbolic grammar which will be applied as test requirements along with the test case generation.

In our approach, each test case produced by GBTG has an associate set of structural testing requirements. Given the symbolic context-free grammar in Figure 1.1, we take an arithmetic expression like Figure 2.1 where each [N] can be substituted by a random integer from its defined domain. This expression has a complete derivation tree as shown Figure 2.2 by applied production rules from grammar, and each complete derivation path is broken into small basic components which describe structural

properties of the test case, from the root node to a leaf node in a coverage tree as shown in Figure 2.3.

$$[N] + [N] - [N] + [N] - [N] - [N]$$

Figure 2.1: Expression Example

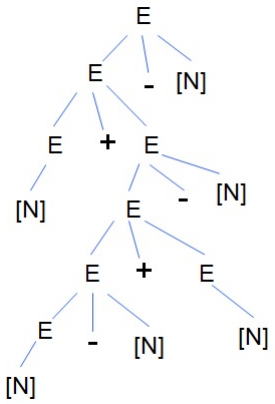

Figure 2.2: Derivation Tree of Expression in Figure 2.1

Now this expression has an associate set of testing requirements {E2E1E0, E2E1E2E0, E0}. Every testing requirement consisting a property sequence of grammar rule indexes from 0 to 2, represents the leftmost derivation sub-tree starting from a leftmost variable E until a terminal symbol is reached at the leftmost leaf while deriving variable E. take first item E2E1E0 from above testing requirements set, it represents a segment of derivations starting from the root, where the leftmost symbol is the variable E. the derivation procedure start from applying the 3-rd production rule of variable E, followed by applying the 2-nd rule of variable E and finally applying the 1-st rule of variable E in sequence, until the leftmost symbol in the derived sequence becomes a terminal [N]. the procedure is show in Figure 2.4

The production procedure of next two testing requirements E2E1E2E0 and E0 are similar to the first one, denoting the segments of derivation routes described in

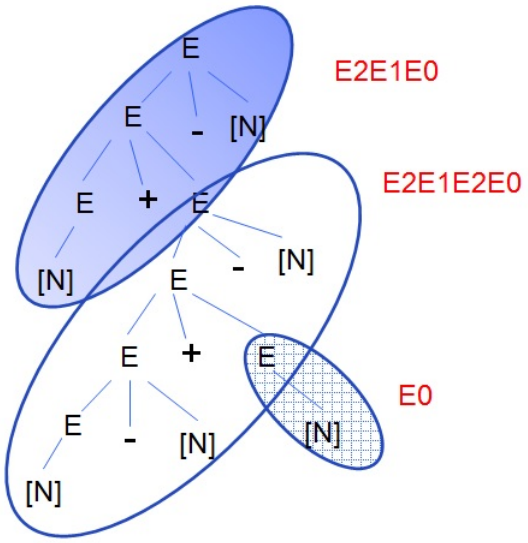

Figure 2.3: Test Requirements: Lefty Subtrees

$$\begin{aligned} E &\Rightarrow E-[N] \\ &\Rightarrow E+E-[N] \\ &\Rightarrow [N]+E-[N] \end{aligned}$$

Figure 2.4: Derivation Procedure of Requirement E2E1E0

Figure 2.5 and Figure 2.6, starting from the left most variable E in Figure 2.4 which is broken point of syntax tree to leftmost sub-trees.

$$\begin{aligned} E-[N] &\Rightarrow E-[N]-[N] \\ &\Rightarrow E+E-[N]-[N] \\ &\Rightarrow E-[N]+E-[N]-[N] \\ &\Rightarrow [N]-[N]+E-[N]-[N] \end{aligned}$$

Figure 2.5: Derivation Procedure of Requirement E2E1E2E0

The standard derivation tree in Figure 2.3 for the generated expression shows that each lefty sub-tree corresponds to one testing requirement actually illustrating

$$E - [N] - [N] \quad \Rightarrow \quad [N] - [N] - [N]$$

Figure 2.6: Derivation Procedure of Requirement E0

a nested sub structure of the whole derivation tree. For a structure-data-input application, for example, software to evaluate arithmetic expressions, the system tester would run as many different expressions as possible to determine that any arithmetic operators and their combinations are supported by the developing system. The nested sub-structures could serve as perfect test coverage criteria for structured input data. Additionally, many certain sub-structures may fall into different testing requirements, which indicate the possibility that testing requirement can imaginable be used for test cases reduction, selection or prioritization.

Chapter 3

Test Suites Reduction

3.1 Test Suites Reduction using Test Requirements

Error detection is a critical task in software testing throughout the whole life cycle of software development. Test suites are often reused while software evolves and upgrades. It yields the result that in a large test suite which should have some redundant test cases considerably, where requirements covered by these test cases are also covered by those other test cases. Re-executing all test cases costs much more resources and time, so that it is important to develop specific techniques to minimize test suite by eliminating redundant test cases, specific techniques to select test cases which are important to software evolution, specific techniques to prioritize test cases that maximize the possibility of fault detection.

Test suite minimization techniques seek to find redundant test cases and to remove them from the test suite The reason of test suites become redundant includes: relationship between input and output is no longer meaningful due to software modifi-

	Requirements					
Test Case	r_1	r_2	r_3	r_4	r_5	r_6
t_1	X	X	X			
t_2	X			X		
t_3		X			X	
t_4			X			X
t_5					X	

Table 3.1: Example test suite taken from Tallam and Gupta [33].

cation, test case generated by a specific program and it changed, structure of software under test is changed.

Chavatal [10] first proposed the usage of a greedy heuristic that chooses test case which covers almost all requirements about to be covered (max cardinality of test set), until all requirements have been satisfied.

Harrold, Gupta, and Soffa [18] have developed another greedy heuristic, based on the cardinality of test case covering specific requirements, to choose a minimal subset of test case which covers the same set of requirements as the original test suite. The idea by Harrold et al. generates the implementations which are always equally good or better than the results computed by Chavatal. On the other side, they still have the worst case execution time of O(|T|*max(|T_i |)) [20]. Here |T| stands for the size of the original test suite, and max(| T_i |) stands for the cardinality of the largest group of test cases in TS.

We apply a similar heuristic technique of the above two methods to minimize a representative set of test cases from a test suite that provides the same coverage as the entire test suite. This minimization is performed by identifying, eliminating the redundant and obsolete test cases in the test suite. The representative set replaces the original test suite and then potentially produces a smaller test suite. Our technique is independent of the testing methodology and only requires an association between

a testing requirement and the test cases that satisfy the requirement. We adopt the method of choosing most constrained feature from constraint logic programming to solve this problem. Most constrained feature method means at each stage of the search, the heuristic technique involves working with the feature that has the least possible number of valid choices. This heuristic is perhaps more clearly understood in relation to the map-coloring problem. It makes sense that, in a situation where a particular country can be given only one color due to the colors that have been assigned to its neighbors, that country be colored next. Applying most constrained feature let us choose the feature that imposes the most constraints on the remain items. Here we first choose the test case which repeated highest frequency in the associated testing sets to the representative sets. Our algorithm then mark the test property of selected test case whose associated testing sets be included that test case and repeat this operation to the final status.

However, a potential issue of the greedy solution is that that the early selection made by the algorithm can be rendered redundant in the end by the test cases subsequently selected and , when a tie situation between multiple test cases happens, one test case is randomly selected. In Table 3.1, the greedy approach will select t_1 first as it satisfied the maximum number of testing requirements, and then continues to select t_2 , t_3 , and t_4 . Unfortunately, after the selection of t_2 , t_3 and t_4 , t_1 becomes redundant. This solution can be useful of the implications among test suites in case to identify which test case became excessive in test suite reduction procedure [33]. Tallam and Gupta propose a new greedy heuristic algorithm called Delayed-Greedy, that is guaranteed to obtain same or more reduction in the test suite size as compared to the classical greedy [10] [18] heuristic.

3.2 Test Reduction Algorithm

3.2.1 HGS Algorithm

The implementation of our approach for test suite reduction with constrained variables is based on Harrold, Gupta, and Soffa [20] **(HGS)** heuristic method of test suite minimization algorithm, so we first give a brief introduction of HGS algorithm as follows:

1. Initial state:

   a) Input data: T_1 , T_2 T_n testing sets for r_1 , r_2 ,r_n respectively.

   b) Output data: RS=empty, a representative set.

   c) All requirements r_1 , r_2 , r_n are unmarked.

2. Second step: For all test cases that is occurred only once in associated testing sets for requirements respectively are selected and put into representative sets RS, then marks all requirements(testing sets) which covered by these test cases.

3. Third step: Consider the unmarked requirements (testing sets) of cardinality two.

   a) If we only have one requirements with cardinality two, the test cases are chosen and put into RS, then unmark the requirements which covered by these test cases.

   b) If we have several requirements with cardinality two, the test case which has maximum number of occurrence of requirements is selected.

   c) If several test cases are tied with maximum number of occurrence, check these test cases in requirements with next successively higher cardinality,

if the maximum cardinality is reached and still remain several tied test cases, and then select one test case arbitrarily from these tied test cases.

4. Fourth step: Repeat the third step, until reach maximum cardinality or all requirements (testing sets) are marked.

We take the example in Table 3.2 to describe the procedure of the HGS algorithm first. Requirements r_2 is the only singleton testing set. Therefore, test case t_5 is selected and added into RS, requirements r_1 and r_2 are marked since they are covered by t_5 . Then, we study unmarked requirements with cardinality two r_4 , r_5 , r_6 . Test case t_3 and t_4 appears once in these requirements, test case t_1 and t_6 occurs two times of these requirements. Now there is a tie between t_1 and t_6 with highest occurrence, we continue check t_1 and t_6 in unmarked requirements with cardinality three, and then r_3 and r_7 are checked next. At this time, we only compute the occurrence of the tied test cases t_1 and t_6 in r_3 and r_7 . The test case t_1 appears in r_3 while test case t_6 has zero appearance in both r_3 and r_7 . So that test case t_1 is selected and put into RS. Requirements r_3 , r_5 and r_6 are marked which covered by test case t_1 . Continue the processing step with unmarked requirement r_4 which only has cardinality two. Now test case t_3 and t_6 is tie that makes us to continue check their occurrence in unmarked requirements with cardinality three. We found that test case t_3 is occurred in r_7 , thus test case t_3 is selected and put into RS, and then remaining requirements r_4 , r_7 , and r_8 are all marked. Thus the minimized representative sets generated by HGS algorithm for this example is r_5 , r_1 , r_3 .

3.2.2 Our Most Constrained Variable Algorithm

Now we describe our Most Constrained Variable Algorithm **(MCV)** of test reduction. It may be useful to select the specific test case first which has most constrained feature

	Requirements							
Test Case	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8
t_1	X		X		X	X		
t_2			X					X
t_3			X	X			X	X
t_4							X	X
t_5	X		X		X			
t_6				X		X		
t_7							X	X

Table 3.2: Requirements criterion coverage information for test cases in T

in the reduced test suites so that we will generate minimized reducted set while still hold same coverage as the entire test suites, optimistically hold default detection effectiveness of test suites.

The main algorithm is described in Figure 3.1

Consider the example in Table 3.2, the test suite TS consists of test cases, t_i , the test requirements, REQ_l or r_l , and the associated testing sets, Tl. The heuristic first computes value $OCCUR_i$ of each test case t_i : (t_1, 4), (t_2, 2), (t_3 , 4), (t_4 , 3), (t_5 ,2), (t_6 ,2), (t_7 ,2); set of $RELATE_i$: (t_1,(1,3,5,6)), (t_2,(3,8)), (t_3 ,(3,4,7,8)), (t_4 ,(5,7,8)), (t_5 ,(1,2)), (t_6 ,(4,6)), (t_7 ,(7,8)).

Test case t_1 and t_3 both has max value of $OCCUR_i$, randomly choose t_1 first and add it to the representative sets. Then mark REQ_1, REQ_3, REQ_5, REQ_6, reduce the value of $OCCUR_i$ respectively: (t_2, 2) to (t_2, 1) , value of $RELATE_i$: (t_2,(3,8)) to (t_2,(8)) because of the REQ_3 is covered. Same adjustment have done to other test cases,

now we have: $OCCUR_i$ (t_1,0), (t_2, 1), (t_3 , 3), (t_4 , 2), (t_5 ,1), (t_6 ,1), (t_7 ,2);

and $RELATE_i$: (t_1,( )), (t_2,(8)), (t_3 ,(4,7,8)), (t_4 ,(7,8)), (t_5 ,(2)), (t_6 ,(4)), (t_7 ,(7,8)).

Algorithm ReduceConstrainVariable

1. Initial Step:
   a) Read input data;
   b) RS = { };
   c) Unmark all requirements;
   d) ReqSize = size of all requirements;
   e) TestcaseSize = size of all test case;
   f) For each requirement do:
      i. testcase.frequency++;
      ii. add requirement to test case covered set;
   g) endfor
2. Second Step:
   a) Compute frequency of each test case;
   b) Record related test case property ri to test case;
   c) Build binary heap based on frequency, root node has maximum frequency;
3. Third Step:
   a) Next_TestCase = SelectedTC;
   b) RS = RS ∪ { SelectedTC };
   c) Update frequency of test case related the requirement which covered by selectedTC;
   d) Mark the requirements which covered by selectedTC;
   e) Remove selectedTC from binary heap;
   f) Maintain binary heap;
4. Loop Step:
   a) While binary heap size > 0 or frequency of selected test case = 0 do
      i. Third Step;
   b) endWhile;

Function selectedTC

1. While heap size > 0 do
2. maxNode = rootNode;
3. Switch rootNode and last node;
4. Size of binary heap reduce 1;
5. Maintain binary heap;
6. Return maxNode;

Figure 3.1: Algorithm for test suite reduction with constrained variables

Repeat step 2, pick up test case t_3 to representative set. After modification we have:

OCCUR$_i$ (t_1,0), (t_2, 0), (t_3 , 0), (t_4 , 0), (t_5 ,1), (t_6 ,0), (t_7 ,0)

and RELATE$_i$: (t_1,( )), (t_2,( )), (t_3 ,( )), (t_4 ,( )), (t_5 ,(2)), (t_6 ,( )), (t_7 ,( )).

Finally we pick up test case t_5 and get the representative sets (t_1,t_3 ,t_5).

3.2.3 Combination with HGS and Global Frequency

During the experiements of these HGS and MCV two techniques, it gives us another idea of implementing HGS approach with global frequency of test case instead of the frequency of test case in the smallest testing set with same cardinality. HGS algorithm is always looking for the selected test case with higest occurence from lower testing sets with same cardinality, it shows the importance of the test case in these handled testing sets. MCV algorithm is likely to persue the test case with highest frequency in the unmarked test suites, no matter it sitts in the testing sets with small cardinality or enmours cardinality.

Now we combine these two properties of HGS and MCV, and developed the third algorithm which illustrated in Figure 3.2

3.3 Experimental Results of Test Suites Reduction

3.3.1 Virtual Input Data

In order to better illustrate the performance of our algorithm, we provide several versions of our virtual input data described in table 3.3. We adopt the fixed number

Algorithm ReduceConstrainVariable

1. Initial Step:
   a) Read input data;
   b) RS = { };
   c) Unmark all requirements;
   d) TestcaseSize = size of all test case;
   e) For each requirement do:
      i. testcase.frequency++;
      ii. add requirement to test case covered set;
   f) endfor
2. Second Step:
   a) Compute frequency of each test case;
   b) Record related test case property ri to test case;
   c) Build binary heap based on frequency, root node has maximum frequency;
3. Third Step:
   a) Next_TestCase = SelectedTC;
   b) RS = RS ∪ { SelectedTC };
   c) Mark the requirements which covered by selectedTC;
   d) Remove selectedTC from binary heap;
   e) Not update frequency of test case related the requirement which covered by selectedTC;
   f) Maintain binary heap;
4. Loop Step:
   a) While binary heap size > 0 or frequency of selected test case = 0 do
      i. Third Step;
   b) endWhile;

Function selectedTC

1. While heap size > 0 do
2. maxNode = rootNode;
3. Find rest node which has same cardinality with the rootNode;
4. Check frequency of test case in these sets; /*not need next search*/
5. tempTC = test case with highest frequency;
6. Size of binary heap redue 1;
7. Maintain binary heap;
8. Return tempTC;

Figure 3.2: Algorithm for test suite reduction with HGS and Global Frequency

Input Data	Max size of test case set per requirement	Max frequency per test case
1		5
2		10
3	(Max test case size	15
4	/ Reqsize) * 4	20
5		25

Table 3.3: Test case input data argument setting

for the max size of testing set per requirement, and selected five groups of data of max frequency per test case.

For each category we construct three groups of input data organization of the size of requirements and test case: (1) requirements size from 25 to 5000 and test case size 5000, (2) requirements size 500 and test case size from 1000 to 9000, (3) requirements size 1000 and test case size from 1000 to 9000.

3.3.2 Experimental results and analysis

For giving the distinction between the based algorithms used by Mary Jean Harrold et al. for test suite minimization, our algorithm with constrained variables, and another technique with combination of based Mary Jean Harrold algorithm and global frequency, we respectively refer them as HGS algorithm, MCV algorithm, and HGF algorithm.

We implement these three algorithms in C programming. We conduct the following experiment using the five categories based on max frequency per test case includes 5, 10, 15, 20, 25. We fixed another important parameter max size of testing set per requirements to four times of the ratio of test case size in test suite to requirement size. The results of our experiment with the above arguments are respectively shown from table 3.4 to table 3.6, the columns shown in these tables labeled Requirements

Index	Input Data		HGS		MCV		HGF	
	Requirements	**TeseCase**	Time	**Size**	Time	**Size**	Time	**Size**
1	500	**1000**	0.024	**191**	0.002	**172**	0.023	**191**
2	500	**2000**	0.033	**192**	0.004	**150**	0.029	**195**
3	500	**3000**	0.049	**187**	0.007	**140**	0.040	**188**
4	500	**5000**	0.074	**188**	0.011	**135**	0.049	**200**
5	500	**6000**	0.096	**185**	0.015	**129**	0.054	**192**
6	500	**7000**	0.112	**182**	0.016	**128**	0.051	**187**
7	500	**8000**	0.117	**179**	0.018	**130**	0.068	**192**
8	500	**9000**	0.151	**181**	0.021	**124**	0.082	**203**

Table 3.4: Experiment results based on TestCase size, Max Frequency 5

(size of requirements of input data), Testcase (size of test case of input data), HGS (Mary Jean Harrold et al. algorithm), MCV (our approach with most constrained variables), HGF (combination of HGS and global frequency).

Before our experiment, we suppose that our approach MCV algorithm will have the more minimal representative set that HGS algorithm according to our algorithm implementation for choosing the most important test case first than the others, and it may have less efficient running time than HGS algorithm.

3.3.2.1 Experiment 1 focus on Test Case Size

In the first category of our experiments, we let Requirement Size and Max Frequency stable, to show the size of representative sets and running time based on the TestCase size changing.

The values reported in the table 3.4 are the running time of the algorithm, and the size of representative sets computed. For comparison with the above two techniques, the original HGS algorithm is also implemented with respect to their requirements criteron coverage. These results are shown for the TestCase Size range 1000-9000 while Requirement size 500 and Max Frequency 5 in table 3.4.

From the Figure 3.3 it shows that the size of minimal set generated by MCV

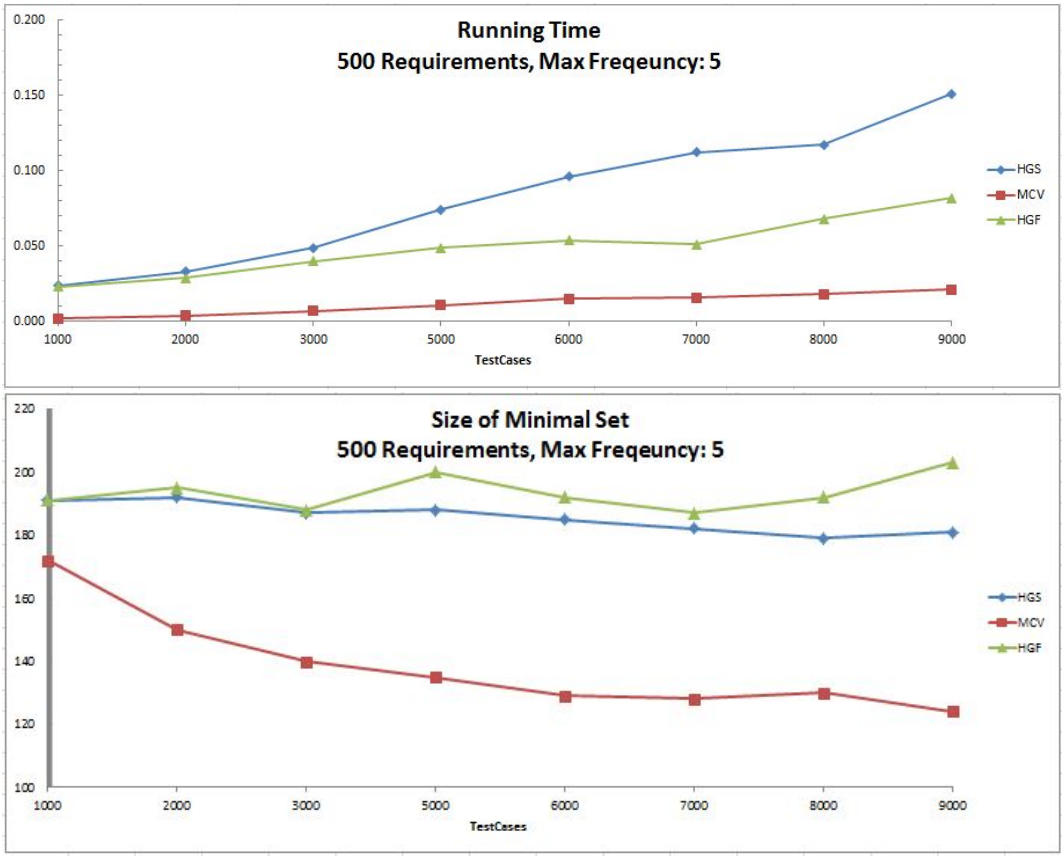

Figure 3.3: Running Time and Minimal Set according to TestCase size changing

algorithm is always smaller than the size of minimal set generated by either HGS algorithm or HGF algorithm at any extent of testcase size in our selected experiments results. The HGF approach gives us a little smaller size of minimal set than HGF algorithm. For the running time, MCV algorithm still presents the better perfromance than other two techniques, and HGF algorithm runns faster than HGS algorithm.

3.3.2.2 Experiment 2 focus on Requirement Size

In the follow category of our experiments, we let TestCase Size and Max Frequency stable to show the results of the size of representative sets and running time based on the Requirements Size changing.

Index	Input Data		HGS		MCV		HGF	
	Requirements	TeseCase	Time	**Size**	Time	**Size**	Time	**Size**
1	**25**	5000	0.034	**6**	0.016	**3**	0.014	**3**
2	**50**	5000	0.020	**7**	0.016	**5**	0.012	**3**
3	**100**	5000	0.028	**17**	0.016	**10**	0.013	**3**
4	**200**	5000	0.029	**30**	0.016	**20**	0.015	**3**
5	**400**	5000	0.048	**58**	0.016	**42**	0.017	**3**
6	**1000**	5000	0.113	**157**	0.015	**115**	0.039	**3**
7	**2000**	5000	0.248	**356**	0.015	**287**	0.104	**3**
8	**4000**	5000	0.697	**801**	0.015	**752**	0.298	**3**

Table 3.5: Experiment results based on Requirement size, Max Frequency 20

The values reported in the table 3.5 are the running time of the algorithm, and the size of representative sets computed. These results are shown for the Requirement Size range 25-4000 while TestCase size 5000 and Max Frequency 20 in table 3.5.

The Figure 3.4 illustrates that the difference of size of minimal set between these three implementations are slightly small when the max frequency of test case reaches 20. And we still yield the smallest size of minimal set among these three techniques. On the other size, neither HGS algorithm nor HGF algorithm attempt to achieve more quick running time than our approach—MCV algorithm.

3.3.2.3 Experiment 3 focus on Max Frequency

In the third category of our experiments, we let TestCase Size and Requirement Size stable to show the results of the size of representative sets and running time based on the Max Frequency changing.

The values reported in the table 3.6 are the running time of the algorithm, and the size of representative sets computed. These results are shown for the Max Frequency range 5-25 while TestCase size 5000 and Requirement size 20 in table 3.6.

Surprisingly note from the Figure 3.5 that when the max frequency reaches half of test suies size(exceedingly rare), the size of minimal set generated by HGF algo-

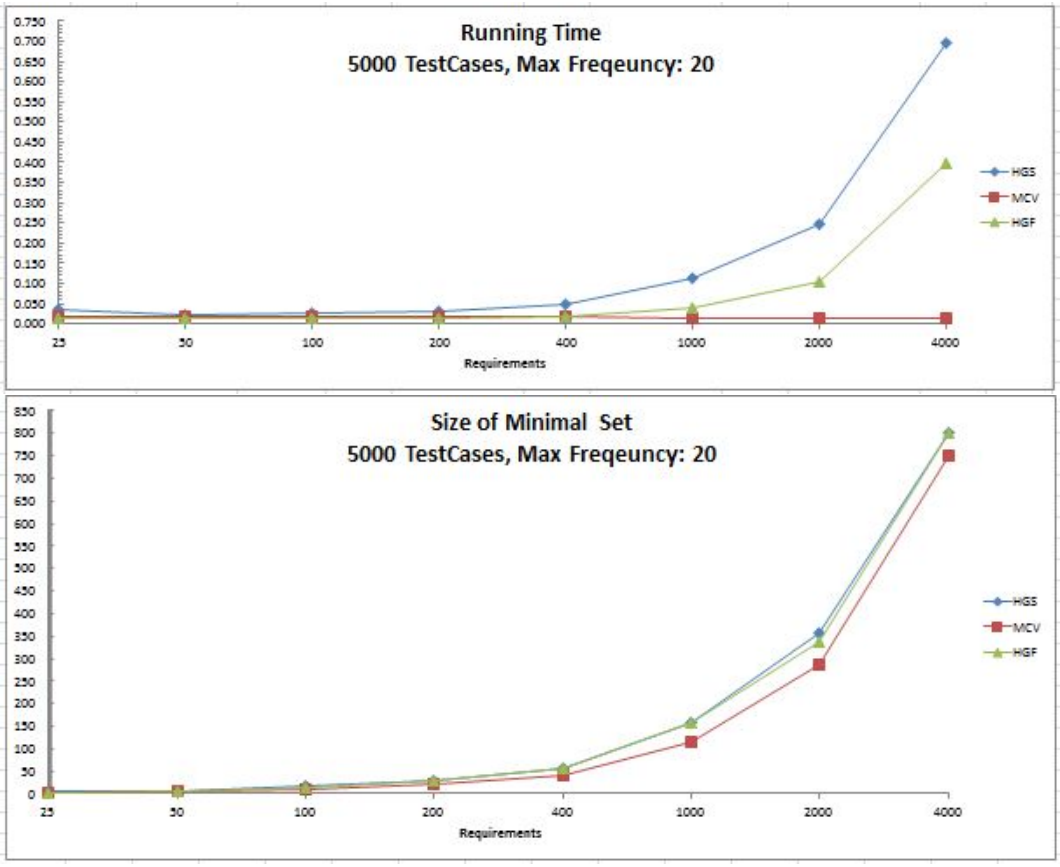

Figure 3.4: Running Time and Minimal Set according to Requirement size changing

Index	Input Data		HGS		MCV		HGF	
	TestCase	**MaxFreq**	Time	**Size**	Time	**Size**	Time	**Size**
1	5000	**5**	0.061	**146**	0.013	**102**	0.034	**158**
2	5000	**10**	0.077	**87**	0.012	**63**	0.033	**104**
3	5000	**15**	0.092	**78**	0.013	**50**	0.024	**69**
4	5000	**20**	0.048	**58**	0.016	**42**	0.017	**58**
5	5000	**25**	0.036	**50**	0.013	**37**	0.021	**53**
6	5000	**1/3 TcSize**	0.019	**27**	0.012	**21**	0.011	**27**
7	5000	**1/2 TcSize**	0.019	**32**	0.012	**20**	0.017	**12**

Table 3.6: Experiment results based on Max Frequency 5, Requirements size 400

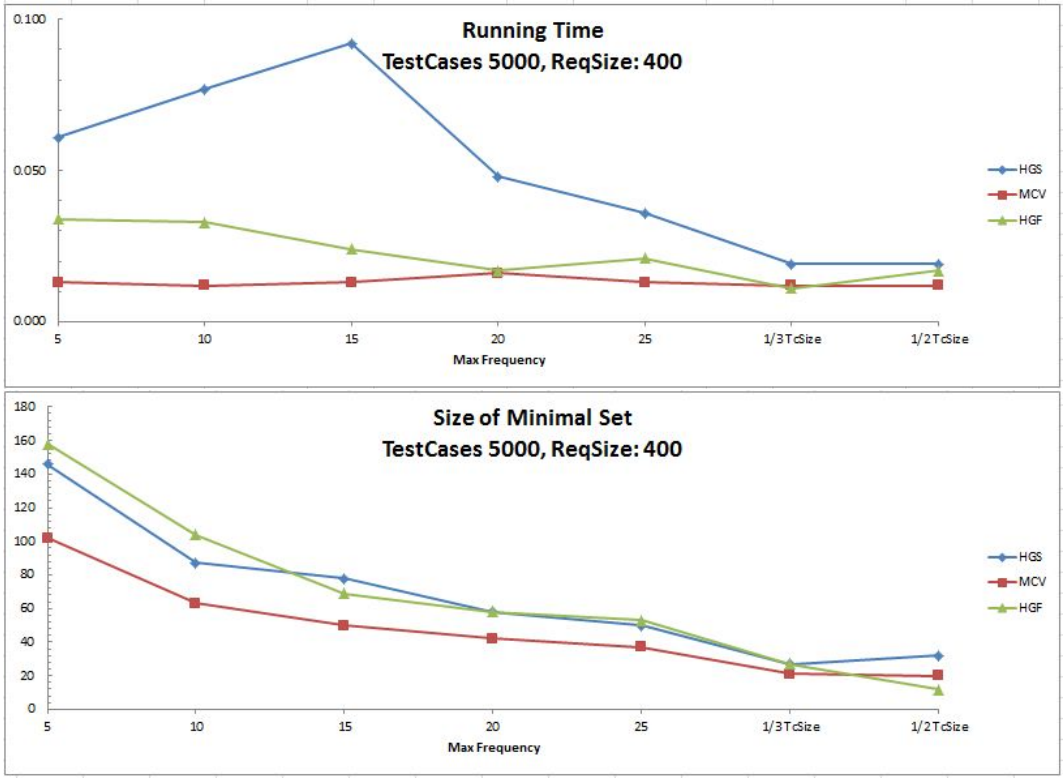

Figure 3.5: Running Time and Minimal Set according to Max Frequency changing

rithm is smaller than MCV algorithm while MCV approach retain gets smaller size of minimal set at other extents. The running time tells us that the approach of MCV algorithm is better than HGS and HGF algorithm.

Now we observe from table 3.4 to table 3.6 that the final size of representative set of test suites minimization with constrained variables generated by MCV algorithm was always smaller than HGSs representative set, these results are expected, at mean while its efficiency of running time is much better than HGSs algorithm. Neither HGS algorithm nor HGF algorithm can reach smaller representative set. However, the efficiency and final representative set from HGS and HGF are not determined which one is better than the other.

Chapter 4

Granularity of Test Requirements

4.1 Related Research

Test suite reduction uses test requirement to determine if the reduced set maintains the original suite's requirement coverage. Based on observations from our previous experimental researchs on test suite reduction discussed in Chapter 3, we suppose there is a need for customized test requirements for fault detection of software. In this section, we examine granularity-based customized test requirements for the fault detection problem. We conduct an extensive experimental study to evaluate the effectiveness between different depth control of test requirement with respect to reduced set size, requirement size, and correct ratio comparision with original test suite.

Each test case generated automatically always comes with a set of test requirements, so that when the test case is used in software testing, we know that what features of a system have been tested. Test requirements of test case from grammar-based generation often have very complicated structure to describe the lefty-subtree of the test case.

Test requirement granularity reflects the way that part property of test require-

ments are taken from it, and are grouped into a new test requirements set. Our objective is to detect the fault of software through test requirements generated along with test case. To do this, we try to obtain test requirement of varying granularities, in a manner that extraction from test requirement with complicated structure might help us to find the fault of software. Our approach is to construct new test requirements of varying granularities by requirement depth. Requirement depth is the flag that determines how far away from the start variable to the stop position for a target test requirement.

For example, if we have a test case t

$$320/567 * (98) + 574$$

with a set of test requirement Rs {E1E0F1F2F0T0, T0, T1, E0F0T0, F0T0 }. We try to use requirement depth to build new test requirement instead of itself. Take requirement E1E0F1F2F0T0 as example, we have a new requirement based on different granularity as shown in 4.1

Depth	Requirement
1	E1
2	E1E0
3	E1E0F1
4	E1E0F1F2
5	E1E0F1F2F0
6	E1E0F1F2F0T0

Table 4.1: Substring as Requirement

4.2 Algorithm of Our Implementation

Our goal of is to build new test requirements of varying granularities by exploring different depth. Take test requirement *Rs* {E1E0F1F2F0T0, T0, T1, E0F0T0, F0T0 } as example, if we choose 1 as requirement depth, then substring of E1E0F1F2F0T0 on depth 1 is E1, substring of E0F0T0 is E0, and substring of F0T0 is F0. So we can get *NRs* {E1, T0, T1, E0, F0 } for next step. If we choose requirement depth as 2, then the *NRs* will be {E1E0, T0, T1, E0F0, F0T0 }, here any requirement whose string length less than **(requirement depth) * 2** will be kept in *NRs*. And so on for requirement depth 3 until highest level.Full description is shown in Figure 4.1

Algorithm for Granularity of Test Requirement

1. First Step:
   a) Input data: test requirement set *Rs*, depth *d*
   b) Output: new test requirement set *NRs* = { };
2. Second Step:
   a) For each test requirement *r* in *Rs* do:
   b) If (length of *r*) <= (d * 2)
      i. If *r* not in *NRs*, put *r* into *NRs*;
      ii. else skip;
      iii. Endif;
   c) break;
   d) Endif
   e) Get *d* depth substring *nr* from *r*;
      i. If *nr* not in *NRs*, put *nr* into *NRs*;
      ii. else skip;
      iii. Endif;
   f) Endfor;

Figure 4.1: Algorithm for Granularity of Test Requirement

4.3 Experimental Results

Depth	Reduced Set	Requirement
1	1	8
2	3	19
3	9	46
4	31	110
5	96	238
6	189	393
7	265	499
none	299	543

Table 4.2: Size of Test Cases and Requirement for each depth

We take test suite (500 test cases) as the input data; extract substring of each test requirement, through depth 1 to 7. None depth level means that we keep whole string itself as test requirement. For each depth extraction we now have different size of test cases and test requirements after test reduction as shown in Figure 4.2. For depth 1, it has 8 requirements which describe 8 basic production rules in symbolic context-free grammar 2 shown in Figure 6.1. We only need 1 test case in reduced set which covers all these 8 test requirements. While we choose more deep depth to control new test requirements generation, it gives us more and more test requirements to describe the feature of test case. At meantime, the size of reduced set is rising. If we do not control the depth of requirements, we may get total 543 test requirements of 299 test cases in reduced set from original test suites with 500 test cases.

Then we feed these 8 groups, reduced set of test cases, into 13 Java program to evaluate the mathmatics results shown in Figure 4.3. The left column is the index of 13 programs under test. The second column shows the correct ratio by feeding total 500 test cases. The right most column gives us the correct ratio by using none depth control to reduce test suite. The rest columns describe the correct ration while using

Program Index	Test Suite 500	Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6	Depth 7	Depth none
1	10.80%	0%	33.33%	2.11%	3.18%	3.10%	1.58%	2.32%	3.34%
2	78.00%	0%	33.33%	66.67%	77.42%	70.83%	76.19%	78.87%	77.10%
3	78.00%	0%	33.33%	66.67%	77.42%	70.83%	76.19%	78.87%	77.10%
4	100.00%	100%	100%	100%	100%	100%	100%	100%	100%
5	1.80%	0%	0%	0%	2.23%	2.08%	1.53%	2.51%	1.67%
6	4.40%	0%	33.33%	0%	0.45%	0.25%	0.41%	0.67%	1.00%
7	100.00%	100%	100%	100%	100%	100%	100%	100%	100.00%
8	8.80%	0%	33.33%	0%	5.68%	4.33%	4.47%	3.25%	3.34%
9	5.40%	0%	33.33%	0%	1.77%	2.38%	1.81%	1.95%	2.01%
10	100.00%	100%	100%	100%	100%	100%	100%	100%	100.00%
11	57.00%	100%	66.67%	66.67%	41.61%	45.21%	46.08%	46.98%	48.83%
12	1.20%	0%	33.33%	0%	1.45%	1.29%	1.49%	1.66%	0.33%
13	10.60%	0%	33.33%	4.11%	4.90%	4.46%	4.64%	4.32%	4.68%

Table 4.3: Ratio of Correctness on Depth Change

reduced set based on different depth control of test requirement. Here we select depth control from depth 1 to depth 7.

For depth 1 and 2, the correct ratio may have significant difference to the original test suite .It is suggesting that building new test requirement in depth 1 or 2 make no sense, it will not help us anymore to detect the fault of software. Generally, correct ratio of reduced set is supposed littler lower than correct ratio of original test suite. However, for program 5, the correct ratio on depth 4, 5, 7, and none depth control is higher than original test suite. Same situation also happens on program 12 for granularity depth control 4, 5, 6, 7. And for program 2 and 3, this phenomenon only happened on depth 7. This is because the new test requirement we extract from original one based on granularity control may be cannot describe enough property of that test case.

Based on our experimental results, we observe that correct ratio is generally arise while the depth of requirement increase. More higher the granularity depth is, more closer the correct ratio are to the original test suite. In this case, in next step fault detection part, we directly use the whole requirement without any depth control to identify the fault of software.

Chapter 5

Automatical Fault Detection in Failed Test Cases

5.1 Automated Fault Detection Approach

Usually, test requirements are commonly used for test case minimization, prioritization etc. Compare to the test requirements from model-based generation, these structured test requirements from grammar-based generation can even more effective and critical to automatically detect the fault of software over the functionality of test case reduction.

Now we feed the reduced test case set as the input to the software under test and evaluate all of reduced test case set. Each failed test case *ft* which does not have the expected results has an associated set of structured test requirements *rs*. Each test requirement r in *rs* which describes a sub-structure of this test case t can be instantiated to a new test case *nt* by an *on-demand* test case generator. The *on-demand* generator can produce a just-enough instant test case *nt* corresponding to the test requirement r, using standard derivation.

For example, the test requirement $r\ E2E1E2E0$ from rs of expression as shown in Figure 2.1 , which describes the leftmost derivation sub-tree starting from a variable E, followed by choice of production rules E2, E1, E2, and E0 in a sequential order. The procedure of derivation of test requirement $r\ E2E1E2E0$ is shown as in Figure 5.1

$$\begin{aligned} E &\overset{E2}{\Longrightarrow} E - [N] \\ &\overset{E1}{\Longrightarrow} E + E - [N] \\ &\overset{E2}{\Longrightarrow} E - [N] + E - [N] \\ &\overset{E0}{\Longrightarrow} [N] - [N] + E - [N] \end{aligned}$$

Figure 5.1: Derivation of On-Demand Generator

With the string [N] - [N] + E - [N] derived from test requirement $r\ E2E1E2E0$, our *on-demand* generator gives us an instant test case *nt* through substituting every [N] with an instance number from the domain we defined in Figure 1.1, and replacing E with a default base case (an instance of [N]).

Since test requirement $r\ E2E1E2E0$ comes from failed test case *ft*, so that the back-produced test case *nt* can be also fed into the software under test to determine whether the associated test requirement $r\ E2E1E2E0$ commits a fault to the failure of the failed test case *ft*. The foundation of our automatic fault detection approach is based on such an important observation to achieve our final goal.

5.2 Finest-Grained Faults Isolation

Although we get the test requirement from the above analysis to determine that this test requirement would be one of the faults of software under test, but this test requirement often too complicate due to the long structure of lefty-sub-tree from

derivation from failed test case *ft*. Can we locate the finer-grained structure of failed test requirement r to identify more precisely issue of the software? The answer is yes.

Assume we have a software under test (SUT): $\mathcal{S}$ and a set of generated test cases $Ts = \{t_1, \cdots, t_n\}$ where each element t of Ts is one test case. We describe a testing procedure which returns a Boolean value indicating whether $\mathcal{S}$ runs correctly given the test case t as $\mathcal{T}(t)$. For simplicity, we hide a parameter $\mathcal{S}$ from the $\mathcal{T}$ procedure and other following procedures as well, assuming $\mathcal{S}$ as a constant global variable. We say t is a *succeeded* test case of $\mathcal{S}$, if $\mathcal{T}(t)$ returns $true$; otherwise t is a *failed* test case of $\mathcal{S}$ if $false$ is returned.

For the set of generated test cases $Ts = \{t_1, \cdots, t_n\}$ and each test case t_i, where $1 \leq i \leq n$, we use $Rs(t_i) = \{r_{i1}, \cdots, r_{im}\}$ to denote its associated set of test requirements. Each test requirement r can be instantiate to a new test case $nt(r)$ of test requirement r generated by the *on-demand* test generator. In this circumstances, for each test case t_i, we can define a set of *faulty test requirements* $FRS(t_i)$, where each requirement $r \in FRS(t_i)$ has a failed instant test case, $nt(r)$, with respect to the software under test, as follows:

$$FRS(t_i) = \{r \in Rs(t_i) \mid \mathcal{T}(nt(r)) = false\}.$$

$FRS(t)$ is empty for a successful test case t, whereas for a failed test case t, $FRS(t)$ may not be null, representing a set of *faults*. We further extend the definition of FRS on a given set of test cases Ts,

$$FRS(Ts) = \bigcup_{t \in Ts} FRS(t).$$

Note that if a test requirement r contributes a fault to the failure of software

testing, we expect $\mathcal{T}(nt(r))$ to be $false$ for any instant test case of r. Practically, the fact may too far from our imagination when some instant test cases could accidently produce expect results. Meanwhile, even for a failed test case t, its set of faulty test requirements $RFS(t)$ may still be empty. This scenario possibly happens because of that test requirements only represent sub-structures (e.g., lefty subtrees) of input data, which may not be plentiful enough to represent some more complicated faults of $\mathcal{S}$. In this special condition, a failed test itself *ft* can be treated as a fine-grained fault of $\mathcal{S}$.

The *faulty test requirements* $FRS(Ts)$ itself is a set of detected faults; and the structures of faulty test requirements can help us to identify the causes of software failure. However, since the depth of left tree could be unbounded, and a test requirement r with a lefty-tree structure, may still be not too short to identify the exact causes of software failure. On the other side, $FRS(Ts)$ may still include too many sub test requirements with various lengths, which is not helpful for us to determine the precise causes of software failure.

To solve this problem, for each $r \in FRS(Ts)$, we use an another independent function, $\mathcal{I}(r)$, undertaking to find a set of *finest-grained* faults. Describe the test requirement r be in a form of $I_1 I_2 \cdots I_k$, where $k \geq 1$; we first define an isolation function with a grain size d, $\mathcal{I}(r, d)$, where $1 \leq d \leq k$, illustrated in Figure 5.2:

$$
\begin{array}{rcl}
\mathcal{I}(r,1) & = & \{\ I_i \mid 1 \leq i \leq k \text{ and } \mathcal{T}(nt(I_i)) = false\ \} \\
\mathcal{I}(r,2) & = & \{\ I_i I_{i+1} \mid 1 \leq i \leq k-1 \\
 & & \qquad \text{and } \mathcal{T}(nt(I_i I_{i+1})) = false\ \} \\
 & \vdots & \\
\mathcal{I}(r,k) & = & \{\ I_1 I_2 \cdots I_k\ \}
\end{array}
$$

Figure 5.2: Finest-Grained Function

For a faulty test requirement r, every $\mathcal{I}(r, d)$ describes a set of fine-grained faults of

software under test $\mathcal{S}$, in the form of r's substrings of length d, each of which has a failed instant test case. Now we can define the *finest-grained faults* with respect to r

$$\mathcal{I}(r) = \mathcal{I}(r, m),$$

where m is the smallest number between 1 and k such that $\mathcal{I}(r, m) \neq \emptyset$. In another word, no finer-grained faults $\mathcal{I}(r, l)$, where $1 \leq l < m$, can be found.

Therefore, given a set of generated test cases Ts, we have its finest-grained faults $\mathcal{F}(Ts)$ defined as follows:

$$\mathcal{F}(Ts) = \bigcup_{r \in FRS(Ts)} \mathcal{I}(r).$$

It is suggesting that for each faulty test requirement r, we could find its corresponding finest-grained faults $\mathcal{I}(r)$ in first step; then the finest-grained faults of Ts is basically an accumulated set of those finest-grained faults for each faulty test requirement $r \in FRS(Ts)$.

5.3 An Motivation Example

Back to the software under test SUT, $\mathcal{S}$, whose input data is in the form of an infix arithmetic expression, then transform the infix string of expression to postfix string, and evaluates the arithmetic calculation results. Assume we have a typical java program $\mathcal{S}$ which mistakenly handles arithmetic operators ($-$ and $+$) in a right-associative implementation instead of left-associative approach. The inputs of arithmetic expression are generated by grammar-based test generation with a given grammar in Example shown in Figure 1.1. Suppose we have a test case t.

$$568 + 253 - 863 + 303 - 942 - 138$$

Obviously from the comparison of arithmetic results comparison, t is a *failed* test case *ft* of $\mathcal{S}$ because of the fact $\mathcal{T}(t) = false$, due to its execution method of right-associativity. At the meantime, we have its associated set of test requirements as

$$Rs(t) = \{E2E1E0,\ E2E1E2E0,\ E0\}$$

Through our on-demand test case generator, we can generate each one test case per test requirements r belongs to *Rs(t)* as shown in Figure 5.3

$$\begin{aligned} nt(E2E1E0) &= \text{"}765 + 243 - 839\text{"} \\ nt(E2E1E2E0) &= \text{"}192 - 98 + 765 - 752\text{"} \\ nt(E0) &= \text{"}258\text{"} \end{aligned}$$

Figure 5.3: New Test Generation from On-demand Generator

The represented back-produced instant test cases from our on-demand generator can be also treated as input data into $\mathcal{S}$ to identify whether each associated test requirement r contributes to the failure of $\mathcal{T}(t)$ in this failed test case *ft*. Since we know that, in advance, its mistakenly implementation of right-associativity instead of left-associativity, we retrieve the set of faulty test requirements of t from failed test case nt as follows:

$$FRS(t) = \{E2E1E0, E2E1E2E0\}$$

For each test requirement $r \in FRS(Ts)$, we need to execute the isolation function $\mathcal{I}(r)$ with different size of grain level. Take the test requirement r $E2E1E2E0$ as an example, we have

$$\mathcal{I}(E2E1E2E0, 1) = \{\}$$

The empty set of fine-grained faults of test requirement $E2E1E2E0$ on level 1, based

on the reason of that $\mathcal{I}(nt(I)) = true$ for each $I \in \{E2, E1, E0\}$. For level 2 of the its set fine-grained faults, we have

$$\mathcal{I}(E2E1E2E0, 2) = \{E1E2\}$$

results set because of that on-demand generated new test case $nt(E1E2)$ and $nt(E2E1)$, respectively, generate instant test cases in form of $[N]-[N]+[N]$ and $[N]+[N]-[N]$, where right associativity only affects the calculation result of the former one. Similarly we have the fine-grained faults set on level 3 and 4 as

$$\begin{aligned} \mathcal{I}(E2E1E2E0, 3) &= \{E2E1E2, E1E2E0\} \\ \mathcal{I}(E2E1E2E0, 4) &= \{E2E1E2E0\} \end{aligned}$$

If we take a further look at the fine-grained faults at different level, we can see that all the faults found in the 3-rd level $\mathcal{I}(E2E1E2E0, 3)$ and 4-th level $\mathcal{I}(E2E1E2E0, 4)$ are actually caused by the finer-grained faults $E1E2$, where the instant new test case nt is in form of $[N] - [N] + [N]$. Therefore, in our practical implementation, the identificaions of the fine-grained fault of 3-rd level $\mathcal{I}(E2E1E2E0, 3)$ and 4-th level $\mathcal{I}(E2E1E2E0, 4)$ can be eliminated from our total fine-grained fault set. In another word, if we find a fine-grained fault at n-th level of test requirement r, then we can skip the (n+1)-th level until the highest level seeking.

Finally, we get our *finest-grained* faults from test requirements r $E2E1E2E0$:

$$\mathcal{F}(E2E1E2E0) = \{E1E2\}$$

5.4 Algorithm of Fault Detection via Grammar-Based Test Generation

In a summary of above analysis, now we propose our novel automatic fault detection algorithm via grammar-based test generation in detailed pseudo code as show in Algorithm 1 and Algorithm 2.

5.4.1 Automated Fault Detection Algorithm

In our creative approach, we define a main function *faultDetector(Ts,Rmap)* illustrated in Algorithm 1, let Ts be a set of test cases from grammar-based test generation and *Rmap* is a mapping function from a failed test case *ft* to a set of test requirements Rs, which returns a set of detected finest-grained faults as F. The local variable F, start from an empty set (line 4), is applied to act in place of a set of finer faults found in the main function *faultDetector(Ts,Rmap)* , which is keeping updated at runtime of analysis procedure (line 13). For each test case $t \in Ts$, if it is a failed test case *ft* (line 6, where $\neg$ is a logical operator of negation), we find faulty test requirements *ftr* (lines 8-11) of each test requirement r, followed by using an on-demand generator to instantiate each *ftr* to a new instance test case $instance_{nt}$, followed by observing whether r contributes more finer faults or even finest fault to the result set F (lines 13-14, where *finestGrainedFaults(r, F)* is defined in Algorithm 2). For a failed test case ft, if we can not to locate any finer faults other than ft, then the failed test case ft itself will have none contribution to the fault of software and be printed out.

Algorithm 1 Automated Fault Detection Algorithm

```
1: [Input] Ts: a set of test cases; Rmap: a map function from a given test case to
   a set of test requirements
2: [Output] a set of finest-grained faults of Ts
3: function FAULTDETECTOR(Ts, Rmap)
4:     F ← ∅                                          ▷ an initial empty set of faults
5:     for all t ∈ Ts do
6:         if ¬T(t) then                                          ▷ a failed test case
7:             faultFoundFlag ← false
8:             for all r ∈ Rmap(t) do
9:                 instant_nt ← nt(r)                          ▷ an instant test case
10:                                    ▷ nt is an on-demand test case generator
11:                if ¬T(instant_nt) then
12:                                ▷ r is a test requirement from failed test case ft
13:                    F ← finestGrainedFaults(r, F)
14:                    faultFoundFlag ← true
15:                end if
16:            end for
17:            if ¬faultFoundFlag then
18:                print t                          ▷ no faulty test requirements of t
19:            end if
20:        end if
21:    end for
22:    return F
23: end function
```

5.4.2 Finest-Grained Faults Isolation Algorithm

Algorithm 2 defines a function *finestGrainedFaults*, which takes inputs r, a faulty test requirement, and F, a set of found finer faults, checks whether r contributes more finer faults into F, and incrementally update F as a return set. For each grain size i from 1 to $|r|$, where $|r|$ denotes the length or r, the algorithm searchs until a non-empty set of faults of r is found (lines 5-22), otherwise if no finer faults of r can be found, r itself will be added into F (lines 23-26). To search for the finest-grained faults, we introduce a variable i to control the grain size from 1 to the length of the test requirement r (line 7); once a non-empty set of finer faults of r is found, the

indicator, *finerFaultFound*, will be set *true*. Predicate *substr(s,r)* is used to check whether s is a substring of r. For each substring s of r with the current grain size, we first check whether s is already in F or there is a finer-grained fault of s already in F (line 9). If either case holds, we skip processing s since finer-grained faults of s have already been recorded in F; otherwise if there is no finer-grained fault of s already found, we produce an instant test case $instant_{tc}$ by using our on-demand test case generator (line 13), followed by feeding $instant_{tc}$ to the SUT for testing. If $instant_{tc}$ is a failed test case (line 14), we update F by first removing any found faults which are coarser-grained than $instant_{tc}$, then adding $instant_c$ into the fault set F (lines 15-16, similarly in lines 24-25).

One main advantage of maintaining a set of found faults, F, is that no fault, as well as its coarser-grained ones, will be processed twice. This is consistent to the strategy of dynamical programming, and makes our algorithm efficient. For our on-demand test case generator tc, we assume that every generated instant test case $tc(r)$, given a same input r, has the same testing behaviors, either $\mathcal{T}(tc(r)) = true$ or $\mathcal{T}(tc(r)) = false$ for any $tc(r)$. However, practically we can only claim that r is a fault if $\mathcal{T}(tc(r))$ is $false$ for some instant test cases; whereas if $\mathcal{T}(tc(r))$ is $true$, we are unable to affirm that r is not a fault. For this reason, we only maintain a set of found faults F; and for those r's whose $\mathcal{T}(tc(r))$ has been *true* before, they may have chances to generate new instant test cases to be tested again on the SUT.

The running time complexity of our fault detection algorithm is $O(N * L_r * L_{tc})$, where N is the size of (failed) test cases, and L_r and L_{tc} are the maximal lengths of a test requirement and a test case, respectively. We disregard the complexity of the procedure of $\mathcal{T}$ for testing the SUT, by simply assuming its time complexity $O(1)$. The time complexity of function *finestGrainedFault*, in Algorithm 2, is $O(L_r^2)$ due to the fact that there are L_r^2 number of different grain-sized substrings to consider in the

worst scenario, given a test case r. The main procedure *faultDetector*, in Algorithm 1, calls the function *finestGrainedFault* at most $N * K$ times, where K is the size of a set of (faulty) test requirements. For a test case tc, its length L_{tc} roughly equals to $K * L_r$ since L_r represents the length of a test requirement.

Algorithm 2 Finest-Grained Faults Isolation Algorithm

```
1: [Input] r: a faulty test requirement; F: a set of found faults
2: [Output] an updated set of faults
3: function FINESTGRAINEDFAULTS(r, F)
4:     len ← |r|                                   ▷ |r| returns the length of r
5:     i ← 1
6:     finerFaultFound ← false
7:     while ¬finerFaultFound and i < len do
8:         for all s s.t. substr(s, r) and |s| is i do
9:             if ∃w ∈ F s.t. substr(w, s) or s ∈ F then
10:                                  ▷ fault s or its finer fault already found
11:                finerFaultFound ← true
12:            else
13:                instant_tc ← tc(s)                ▷ an instant test case
14:                if ¬T(instant_tc) then
15:                    F ← F − {w ∈ F | substr(s, w)}
16:                    F ← F ∪ {s}                          ▷ a finer fault
17:                    finerFaultFound ← true
18:                end if
19:            end if
20:        end for
21:        i ← i + 1
22:    end while
23:    if ¬finerFaultFound then
24:        F ← F − {w ∈ F | substr(r, w)}
25:        F ← F ∪ {r}
26:    end if
27:    return F
28: end function
```

Chapter 6

Experimental Study

6.1 A Grading System

An automatic grading system for Java programs was provided, using grammar-based test case generation as input data. The Java programming assignment from grading system which takes an infix arithmetic expression as an input string, converts the input infix string into a postfix expression by executing stack operation, and finally a result is returned by calculating the postfix expression. Assuming we have a correct program to evaluate the expected results for each generated expression, and compares the result with the one return from each assignment.

Table 6.1 shows the grading results on 13 Java program submissions, where the middle column is correctness ratios returned from grading system using grammar-based test generation 500 test cases as input, in another word feeding 500 different arithmetic expressions into java program. The right column is the grading results on 50 test cases which designed manually. Although the results from Table 6.1 show that the grammar-based grading system may provide significantly different correctness ratios, but interestingly, it reduces tester the time and cost of constructing all test

Java Program	*Grammar-Based Generation*	*Manual Generation*
1	10.80%	60%
2	78.00%	52%
3	78.00%	52%
4	98.40%	100%
5	1.80%	0%
6	4.40%	0%
7	100.00%	76.00%
8	8.80%	0%
9	5.40%	0%
10	100.00%	100.00%
11	57.00%	62.00%
12	1.20%	0%
13	10.60%	0%

Table 6.1: Ration of Correctness Compare :Automatic and Manual

cases and no need to concerns the coverage and balance of test case generation.

6.2 Test Case Reduction

Test Suites	Test Requirements	Reduced Set	Reduced Ratio
100	203	89	11.00%
200	327	149	25.50%
300	432	217	27.67%
400	505	251	37.25%
500	543	299	40.20%
1000	884	506	49.40%

Table 6.2: Test Suites Reduction Results

Table 6.2 shows that each group of test suites has reduced from the original test suites range from 11.00% to 40.20% while not losing any test requirements covered by original test suites. It is very useful to help software testers to save lots of time and effort to evaluate the software without concerning any loss of test requirements that the system should be covered.

Grammar-based test generation not only free testers from manually design test cases, but also can help us to reduce the test case set with its associate set of test requirements. We implemented a test case reduction algorithm based on the greedy heuristic in [10].The minimization algorithm takes a set of 500 test cases with 543 test case requirements, generated by grammar-based test generation, and yields a reduced set of test cases of size 299.

Java program Assignments	Original Test Case Set Size: 500	Reduced Test Case Set Size: 299
1	10.80%	3.34%
2	78.00%	77.10%
3	78.00%	77.10%
4	98.40%	97.99%
5	1.80%	1.67%
6	4.40%	1.00%
7	100.00%	100.00%
8	8.80%	3.34%
9	5.40%	2.01%
10	100.00%	100.00%
11	57.00%	48.83%
12	1.20%	0.33%
13	10.60%	4.68%

Table 6.3: Ratio of Correctness on a Reduced Set 500

Table 6.3 shows the grading results on both original set of 500 test cases and its reduced set. Except the extreme cases, Programs 7 and 10, the rest correctness ratios on the reduced set are consistently lower than those on the original set. This observation is quite reasonable since a reduced set, produced by a greedy algorithm with a heuristic strategy trying to pick up a test case covering as many test case requirements as possible, usually selects longer expressions. Such a longer expression, with more test requirements, becomes much easier to fail a testing since one buggy requirement is enough to fail the test case. At the same time, the total size of test

cases has been reduced; therefore, the correctness ratio becomes relatively lower on the reduced set.

Java program Assignments	Original Test Case Set Size: 1000	Reduced Test Case Set Size: 506
1	14.00%	6.13%
2	77.90%	77.08%
3	77.90%	77.08%
4	97.50%	96.44%
5	0.60%	0.40%
6	5.30%	2.57%
7	100.00%	100.00%
8	8.80%	2.96%
9	7.10%	2.37%
10	100.00%	100.00%
11	51.80%	42.69%
12	1.30%	0.79%
13	10.80%	4.94%

Table 6.4: Ratio of Correctness on a Reduced Set 1000

Table 6.4 shows the grading results on both original set of 1000 test cases and its reduced set with 884 test case requirements. Except the extreme cases, Programs 7 and 10, same situation as with 543 test case requirements, the rest correctness ratios on the on the original are consistently higher than those reduced set.

The experimental results in Table 6.4 also show consistent grading results, even though slightly lower, between original test cases and their reduced set. It justifies the effectiveness and usefulness of those associated test case requirements.

6.3 Fault Detection via Failed Test Cases

Our experiment uses grammar-based test generation to produce the test case suite while each test case has an associate set of test requirements in the form of arithmetic

expression consists of different combination of arithmetic operator { +, -, *, /, () } and integer numbers [N] from defined domain. The symbolic context-free grammar of arithmetic expressions is described in Figure 6.1

$$
\begin{array}{rcl}
E & ::= & F \quad | \; E \; + \; F \; | \; E - F \\
F & ::= & T \quad | \; F \; * \; T \; | \; F \; / \; T \\
T & ::= & [N] \; | \; (\; E \;) \\
[N] & ::= & 1 \; .. \; 1000
\end{array}
$$

Figure 6.1: Symbolic Context-free Grammar 2

The whole procedure from generate test case based on grammar approach to identify the fault of software under test is briefly summarized as follow:

1. Field: data intensive, for example student homework grading
2. Grammar-based test case generation, each test case (math expression) is generated along with a set of requirements property of math expression, requirements set is a set of left tree structure consists of the combination of operators
3. Test case reduction
    a) Using requirements coverage as standard to minimize the test case
        i. We use greedy approach to implement test case minimization, greedy approach is heuristics, so that the representative sets of greedy minimization is not optimal small, but approximately small
        ii. Results show that the correct rate of the test case set after minimization is close to the original correct rate but a little be lower
        iii. The reason of the lower correct rate: greedy algorithm will always keep the more complicated property rather than short property, and lead to the wrong rate increase

iv. We assume we already have a correct program to generate the standard answer of program

b) Evaluate these test cases(math expressions) after minimization, remove those test cases which has correct math results, and keep the test cases which have wrong results

4. Fault Detection idenfity the bugs of software under test

a) Build a new total requirement array (properties of each test case which has wrong results, these test cases are already under minimization)

i. If it is a substring(include equal) of exist item, skip

ii. If exist item is a substring of it, replace exist item with it.

iii. Generate one Math expression based on each requirement, after evaluation in JAVA, remove those requirements which have right results, keep the requirements which have wrong results.

iv. Now it is the subset requirements of total requirements array

b) For each requirement Ri in subset requirements

i. Analysis first level(single operator) of Ri, generate test cases

ii. Analysis second level (two operators) of Ri, generate test cases

iii. Until highest level(include whole requirement) of Ri, generate test caes

c) For each requirement Ri in subset requirements, evaluate the test cases generated in Step 2

i. Traverse first level, if find error and this operator is not in FinalOuput array, then put related operator in FinalOutput array, go through each one in this level

ii. If find error in step 3.a, skip the rest level of this requirement

iii. If not find error in step 3.a, go to next level until highest level. In each level, we should go through all the items in this level

d) FinalOuput array is our results

We take two groups of grammar-based test generation in size 500 and 1000 in our experiment. For 1000 test cases group data, it has total 884 test requirements as associate set of test cases. The size of reduced set of 1000 test cases is 506.

The experimental results in Table 6.5 are using another test suites with 500 test cases along with 543 test requirements then reduced to 299 test cases which are fed into our 13 java program. In spite of such large set of test requirements, and maybe each test requirement could have a very long structure, we can still isolate the quite specific with very short structure of finest-grained fault of software under test.

From Table 6.5 we can clearly observe the information for identifying the causes of program failure. The set of finest-grained faults of program 1 and program 13 shows that these typical issues are related with handling arithmetic expressions to explain that left-associativity of the task demand are not well processed, that lead to the wrong computation of the expression results in those test cases have at least two mathematic operators including ' - ' and ' / ' . And program 2 and 3 have same problem that as long as the parentheses operator is part of the expression then it will fail the program. For program 5 and program 12 almost fail all the test cases that it is not working at all, even though a single number is fed to them. Program 6, 8 and 9 have similar execution behaviors which implement the program as right-associativity instead of left-associativity and ignore the math operator precedence. However, program 11 only has a specific fault patter as

$$[N] * [N]/[N]$$

Index	Test Suites Ratio	Reduced Set Ratio	Finest-Grained Faults	Causes of Failure
1	10.80%	3.34%	{-+, //, /*, - -, */}	right-associativity
2	78.00%	77.10%	{(}	parenthesis not properly handled
3	78.00%	77.10%	{(}	parenthesis not properly handled
4	100.00%	100.00%	{}	no fault found
5	1.80%	1.67%	{+, -, /, (, *, [N]}	not working at all
6	4.40%	1.00%	{*+, //, /*, - -, *-, */, /+, /-}	right-associativity; operator precedence ignorance
7	100.00%	100.00%	{}	no fault found
8	8.80%	3.34%	{-+, //, /*, *+, - -, *-, */, /+, /-}	right-associativity; operator precedence ignorance
9	5.40%	2.01%	{*+, //, /*, - -, *-, */, /+, /-}	right-associativity; operator precedence ignorance
10	100.00%	100.00%	{}	no fault found
11	57.00%	48.83%	{*/}	$[N] * [N]/[N]$
12	1.20%	0.33%	{+, -, /, (, *, [N]}	not working at all
13	10.60%	4.68%	{-+, //, /*, - -, */}	right-associativity

Table 6.5: Finest-grained Faults and Interpretations

. Interestingly, although program 6, 8 and 9 have similar patterns of the issues, but the program 8 has a little higher correct ration may not necessarily have the smaller set of finest-grained faults than the other two programs, the reason is that in our practical experiment, each pattern of fault only illustrates there is just a failed instance of pattern.

However, our approach of automatic fault detection via grammar-based test generation may not always to locate the shortest structure of program faults. For instance, we could design a long structure of defect pattern as given a pair of parentheses followed by at least one ' * ' math operand and then followed by at least two ' + ' and ' - ' math operands, and put it into one correct program then the program returns a

wrong result. Our methods of fault detection, using a set of grammar-based generated test case, has the ability to find a set of fault patterns, as each pattern contains ' (**+- ', or ' (**++- ' etc. property illustrate the consistency to the designed fault.

Chapter 7

Related Work

7.1 Automatic Test Case Generation

Automatic test case generation is trying to find an input data that will be drive execution of software under test (SUT) along a typical path in the control flow picture. In recent years, automatic test case generation has turned into more essential part of software testing, which frees the testers significant time and effort in software development and maintenance.

W Krenn et al. [24] extend on the formalism of objected-oriented action systems (OOAS) and describe a mapping of a selected UML-subset to OOAS by choosing one of the several possible semantics of Unified Modeling Language (UML) Using a model checker to generate test cases can be very straightforward in model-based development, where we have an executable specification for the software that is in, or is easily translate to, the language of a model checker.

PE. Ammann et al. [6] apply a model checker to the problem of test generation using a new application of mutation analysis, where the test case generation is automatic and each counterexample is a complete test case, and equivalent mutant

identifications is also automatic in a sharp contrast to program-based mutation analysis. A Gargantini et al. [14] propose a specification-based method for constructing a suite of test sequences, where a test sequence is a sequence of inputs and outputs for testing a software implementation.

HS Hong et at. [19] present a theory of test coverage and generation from specifications written in extended finite state machines and describe a method for automatic test generation which employs the capability of model checkers to construct counterexamples. J Offutt et al. [30] take general criteria which include techniques for generating tests at several levels of abstraction for specifications for generating test inputs from stated-based specifications. L Tan et al. [34] consider the specification-based testing in which the requirement is given in the linear temporal logic where its property must be hold on all the executions of the system, which are often infinite in size and length.

In our project, we take the test suites from the grammar-based test generation as our automatic test case generator and feed these test cases where each test case has an associate set of test requirements into the software under test to identify the finest-grained fault of (SUT).

7.2 Automatic Fault Detection

Given a program and a failed test case which fails the software under test, fault detection is seeking to identify the causes of failure including the location, type, and possible fixes for the failure. Lots of researches in this field are falling into two parts: statistical approaches to defect localization and experimental approaches rely on analysis of execution traces.

7.2.1 Statistical Analysis Approach

JA Jones et al. [22] propose an interesting technique that uses visualization to assist with locating the errors or faults of software. Their technique uses color to visually map the participation of each program statement in the outcome of the execution of the program with a test suite.

B Liblit et al. [25] present a low-overhead sampling infrastructure for gathering information from the executions experienced by a programs user community. Then statistical modeling based on logistic regression allows us to identify program behaviors that are strongly correlated with failure. In [26] they design a statistical debugging algorithm that isolates issues in programs containing multiple undiagnosed bugs.

C Liu et al. [28] apply another statistical model-based approach to identify software faults through modeling evaluation patterns of predicates in both correct and incorrect test case running respectively and locates a predicate as fault-correlated. R Abreu et al. [5] have a research on dynamic modeling approach to fault localization which is based on logic reasoning over program traces.

As recent research has shown that software developers are not willing to go through such long lists of unrelated potential fault locations while they may be only have very little chances to reach the final fault of software.

7.2.2 Experimental Analysis Approach

A Zeller [37] discusses that delta debugging works the better the smaller the differences are. The differences between the old and the new configuration can provide a good starting point in finding the faults of software. In [38] he shows that how the delta debugging algorithm how to isolate the relevant variables and values by systematically

narrowing the state difference between a passing run and a failing run which helps us automatically reveal the cause-effect chain of the failure.

M Burger et al. [8] take failed test case into account that they record and minimize the interaction between object to the set of calls relevant for the failure of software. Our creative approach of automatic default detection complements these existing experimental approach techniques by systematically narrowing down the test input space to the simplest grammar instantiations which trigger the failure, it is suggesting that we can tremendously reduce the traces to examine.

Our creative approach of automatic default detection complements these existing experimental approach techniques by systematically narrowing down the test input space to the simplest grammar instantiations which trigger the failure, it is suggesting that we can tremendously reduce the traces to examine.

7.3 Code Coverage Test via Pex and Moles

It has been a tacit assumption that software test coverage criteria should use internal information from either specification or program code. For example, code coverage analysis [29] checks whether the system under test has been thoroughly reached at all statements, all branches or even all execution paths.

We try this approach to apply typical white box analysis tool Pex to perform systematic source code analysis, in a framework Moles which creates delegate-based test stubs. Pex explores the procedures of a parameterized unit test using a technique named dynamic symbolic execution.

Pex [35] is a testing tool that performs systematic code analysis, hunts for boundary conditions and flags exceptions and assertion failures. Moles [11] is a framework for creating delegate-based test stubs and detours in .NET Framework applications.

In general, all program analysis techniques are included the following two branches:

- Static analysis techniques: verify that a property holds on all execution paths based on source code
- Dynamic analysis techniques: verify that a property holds on some execution paths

It cannot detect bugs correctly when applying only static analysis or employing a testing technique that is not aware of the structure of the source code.

Pex is a white box analysis tool which balance information about how one software system is implemented on order to validate or falsify certain properties. Pex implements white box test input generation technique that is based on the concept of symbolic execution to achieve the final goal to automatically and systematically produce the minimal set of actual parameters needed to execute a finite number of finite paths.

However, applying symbolic execution to a real-world program is problematic because of such a programs interaction with a stately environment cannot be forked. Pex explores the procedures of a parameterized unit test using a technique named dynamic symbolic execution which consists of the following steps:

1. Starting with very simple inputs
2. While performing a single-path symbolic execution to collect symbolic constraints on the inputs obtained from predicates in branch statements along the execution
3. Using a constraint solver-Z3-to infer variants of the previous inputs in order to steer future program executions along alternative program paths

```
[PexMethod(MaxRuns = 1000), PexAllowedException(typeof(NullReferenceException))]
public static List<object> infix2Postfix(List<object> infix)
{
    List<object> postfix = new List<object>();
    Stack<string> opStack = new Stack<string>();

    for (int i = 0; i < infix.Count; i++)
    {
        if (isInt(infix[i]))
        {
            postfix.Add(infix[i]);
        }

        else if ((infix[i].Equals("(")))
        {
            opStack.Push(infix[i].ToString());
        }
```

Figure 7.1: C# Source Code from Java Program 2

	infix	result	Summary/Exception	Error Message
8	{")"}		InvalidOperationException	Stack empty.
2..	{Count=28...		InvalidOperationException	Stack empty.

Details:

```
List<object> list;
List<object> list1;
object[] os = new object[28];
os[20] = "";
object s0 = new object();
os[21] = s0;
os[22] = "";
object s1 = new object();
os[23] = s1;
os[24] = "+";
os[25] = ")";
os[26] = ")";
os[27] = ")";
list = new List<object>((IEnumerable<object>)os);
list1 = global::InfixArith.infix2Postfix(list);
}
```

Figure 7.2: Pex Running Result of Java Program 2

Now we change the JAVA code (take Pro 2 as example) to C# code as shown in Figure 7.1 :

We use PexAllowedException to allow the null reference exception of list object. After run Pex explorations, we will get two exceptions as shown in Figure 7.2:

This outputs show that the program will reach the Stack empty exception whenever the input is ') ' or consists of several ') '. And this results is consistent to the fact that the program of student 2 has issues with ' ( ) '.

Chapter 8

Conclusion and Future Work

we present a novel fault detection technique for structured input data which can be represented by a grammar. This method can be applied in testing data-input-critical software. We illustrate that test requirements coming from structured data can be effectively used as coverage criteria to reduce the test suites. We then propose an automatic fault detection approach to identify software bugs which have been shown in failed test cases. Through our approach we can identify specific faults with very short structure of finest-grained fault of software under test.

Furthermore, not all the patterns of fault could be detected by our approach. Fox example, if we design a fault pattern as

$$[N] + [N] * [N]$$

into a program, given a test case

$$t = 235 + 549 * 176$$

our testing procedure is able to find that t is a failed test case, but the fault detection

part may not be able to identify the pattern of fault finer that t itself. The reason of this situation is that our test requirements correspond to liner lefty derivation subtrees; by using left-most derivation, the terminal symbol, ' + ' always terminates the test requirement corresponding to ' 235+ ', and the second terminal symbol ' * ' corresponding to ' 549* ' will be represented in a second test requirement. Same scenario is happened to the terminal symbol as parentheses.

Up to the present, our implementation of fault detection is applied to the arithmetic evaluation system; in the future we will try to expand our approach to large software system using structured data as input, such as compilers or translators. We will also continue to explore alter strategies for producing test requirements and compare their test coverage based on fault detection performance in our future.

Bibliography

[1] Coverity static analysis verification engine. `http://www.coverity.com/products/coverity-save.html`.

[2] Find bugs in java programs. `http://findbugs.sourceforge.net`.

[3] Hp fortify static code analyzer. `http://www.hpenterprisesecurity.com/products/hp-fortify-software-security-center/hp-fortify-static-code-analyzer/`.

[4] Valgrind instrumentation framework. `http://www.valgrind.org`.

[5] R. Abreu, P. Zoeteweij, and A. J. van Gemund. An observation-based model for fault localization. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 64–70. ACM, 2008.

[6] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Formal Engineering Methods, 1998. Proceedings. Second International Conference on*, pages 46–54. IEEE, 1998.

[7] B. Beizer. *Software testing techniques*. Dreamtech Press, 2002.

[8] M. Burger and A. Zeller. Minimizing reproduction of software failures. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 221–231. ACM, 2011.

[9] X. Cai and M. R. Lyu. The effect of code coverage on fault detection under different testing profiles. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[10] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[11] J. de Halleux and N. Tillmann. Moles: tool-assisted environment isolation with closures. In *Objects, Models, Components, Patterns*, pages 253–270. Springer, 2010.

[12] G. Di Fatta, S. Leue, and E. Stegantova. Discriminative pattern mining in software fault detection. In *Proceedings of the 3rd international workshop on Software quality assurance*, pages 62–69. ACM, 2006.

[13] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *Software Engineering, IEEE Transactions on*, 14(10):1483–1498, 1988.

[14] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Software EngineeringESEC/FSE99*, pages 146–162. Springer, 1999.

[15] D. Hao, Y. Pan, L. Zhang, W. Zhao, H. Mei, and J. Sun. A similarity-aware approach to testing based fault localization. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 291–294. ACM, 2005.

[16] M. Harrold and P. Kolte. Combat: A compiler based data flow testing system. In *Proceedings of Pacific Northwest Quality Assurance(Oct.) Lawrence and Craig*, pages 311–323, 1992.

[17] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):270–285, 1993.

[18] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993.

[19] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 327–341. Springer, 2002.

[20] D. Jeffrey and N. Gupta. Test suite reduction with selective redundancy. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 549–558, 2005.

[21] D. Jeffrey and N. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *Software Engineering, IEEE Transactions on*, 33(2):108–123, 2007.

[22] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477. ACM, 2002.

[23] H. Kelly J, V. Dan S, C. John J, and R. Leanna K. A practical tutorial on modified condition/decision coverage. 2001.

[24] W. Krenn, R. Schlick, and B. K. Aichernig. Mapping uml to labeled transition systems for test-case generation. In *Formal Methods for Components and Objects*, pages 186–207. Springer, 2010.

[25] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *ACM SIGPLAN Notices*, volume 38, pages 141–154. ACM, 2003.

[26] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. *ACM SIGPLAN Notices*, 40(6):15–26, 2005.

[27] C. Liu and J. Han. Failure proximity: a fault localization-based approach. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 46–56. ACM, 2006.

[28] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295, 2005.

[29] J. C. Miller and C. J. Maloney. Systematic mistake analysis of digital computer programs. *Communications of the ACM*, 6(2):58–63, 1963.

[30] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, 2003.

[31] J. Radatz, A. Geraci, and F. Katki. Ieee standard glossary of software engineering terminology. *IEEE Std*, 610121990:121990, 1990.

[32] P. R. Srivastva, K. Kumar, and G. Raghurama. Test case prioritization based on requirements and risk factors. *ACM SIGSOFT Software Engineering Notes*, 33(4):7, 2008.

[33] S. Tallam and N. Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 35–42. ACM, 2005.

[34] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *Information Reuse and Integration, 2004. IRI 2004. Proceedings of the 2004 IEEE International Conference on*, pages 493–498. IEEE, 2004.

[35] N. Tillmann and J. De Halleux. Pex–white box test generation for. net. In *Tests and Proofs*, pages 134–153. Springer, 2008.

[36] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[37] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Software EngineeringESEC/FSE99*, pages 253–267. Springer, 1999.

[38] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.

[39] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.

[40] H. Zhu, P. A. Hall, and J. H. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.