ADVANCED COMBINATORIAL TESTING

ALGORITHMS AND APPLICATIONS

by

LINBIN YU

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2013

Acknowledgements

I would like to thank lots of people for their help and encouragement in the past few years. First I want to thank my supervisor Dr. Yu Lei for his thoughtful and instructive mentoring in my research. I cannot finish my PhD work without his support, help and encouragement. I would like to thank all my committee members, including Hao Che, Christoph Csallner and Huang Heng for their support. I would also like to thank all faculties and staffs who ever taught or helped me in the past years for their dedication to the profession.

I would like to thank my fellow doctoral students and other friends in UTA for their generous support and friendship. I would like to thank my wife, my unborn child, and my parents. Without their support, I cannot finish this dissertation. At last, I want to thank all researchers in the references for their creative idea and remarkable results.

July 10, 2013

Abstract

ADVANCED COMBINATORIAL TESTING

ALGORITHMS AND APPLICATIONS

Linbin Yu, PhD

The University of Texas at Arlington, 2013

Supervising Professor: Yu Lei

Combinatorial testing (CT) has been shown to be a very effective testing strategy. Given a system with n parameters, t-way combinatorial testing, where t is typically much smaller than n, requires that all t-way combinations, i.e., all combinations involving any t parameter values, be covered by at least one test. This dissertation focuses on two important problems in combinatorial testing, including constrained test generation and combinatorial sequence testing.

For the first problem, we focus on constraint handling during combinatorial test generation. Constraints over input parameters are restrictions that must be satisfied in order for a test to be valid. Constraints can be handled either using constraint solving or using forbidden tuples. An efficient algorithm is proposed for constrained test generation using constraint solving. The proposed algorithm extends an existing combinatorial test generation algorithm that does not handle constraints, and includes several optimizations to improve the performance of constraint handling. Experimental results on both synthesized and real-life systems demonstrate the effectiveness of the propose algorithm and optimizations.

For the second problem, the domain of t-way testing is expanded from test data generation to test sequence generation. Many programs exhibit sequence-related behaviors. We first formally define the system model and coverage for t-way combinatorial sequence testing, and then propose four algorithms for test sequence generation. These algorithms have their own advantages and disadvantages, and can be used for different purposes and in different situations. We have developed a prototype tool that applies t-way sequence testing on Antidote, which is a healthcare data exchange protocol stack. Experimental results suggest that t-way sequence testing can be an effective approach for testing communication protocol implementations.

Table of Contents

Acknowledgements iii

Abstract iv

List of Illustrations ix

List of Tables xi

Chapter 1 Introduction 1

Chapter 2 Constrained Test Generation 4

2.1 Preliminaries 6

2.2 The IPOG Algorithm 8

2.3 The IPOG-C Algorithm 10

2.3.1 The Base Version of Algorithm IPOG-C 10

2.3.2 Validity Check 12

2.3.3 Optimizations 12

A. Avoiding Unnecessary Validity Checks of Target Combinations 12

B. Checking Relevant Constraints Only 13

C. Recording the Solving History 15

2.3.4 Applying Optimizations to Other Algorithms 16

2.4 Experiments 17

2.4.1 Subject Systems 18

2.4.2 Evaluation of the Optimizations 20

2.4.3 Evaluation of Different Factors 23

A. Test Strength 23

B. Number of Parameters 24

C. Domain Size 24

D. Number of Forbidden Tuples 25

2.4.4 Comparison with Other Tools 26
2.5 Related Work 29
Chapter 3 Combinatorial Sequence Testing 33
3.1 Preliminaries 34
3.1.1 System Model 34
3.1.2 T-way Sequence Coverage 37
3.2 Target Sequence Generation 39
3.3 Test Sequence Generation 41
3.3.1 A Target-Oriented Algorithm 42
3.3.2 A Brute Force Algorithm 44
3.3.3 An Incremental Extension Algorithm 45
3.3.4 An SCC-Base Algorithm 49
A. Build Acyclic LTS 49
B. Find Abstract Paths 50
C. Generate Test Sequences 51
3.3.5 Comparison of Test Generation Algorithms 53
3.4 Experiments 53
3.4.1 Case Study: The Java Threads System 54
3.4.2 Synthesized Systems 55
3.4.3 Results and discussions 55
3.5 Related Work 58
3.5.1 Combinatorial Test Data Generation 59
3.5.2 Test Sequence Generation 59
3.5.3 Test Sequence Coverage Criteria 60
Chapter 4 Case Study: Testing IEEE 11073 PHD 61

4.1 IEEE 11073 PHD STANDARDS 62
4.1.1 Agent and Manager 62
4.1.2 Architecture 64
4.1.3 IEEE 11073-20601 64
4.2 The General Conformance Testing Framework 67
4.2.1 Test Sequence Generator 68
4.2.2 Test Data Generator 69
4.2.3 Test Executor 70
4.2.4 Test Evaluator 71
4.3 A Prototype Tool 71
4.4 Preliminary Results 74
4.5 Related Work 76
Chapter 5 Conclusion and Future Work 78
References 81
Biographical Information 87

List of Illustrations

Figure 1 Illustration of the IPOG Algorithm 9
Figure 2 The base version of the IPOG-C algorithm 11
Figure 3 An example CSP problem 12
Figure 4 Illustration of Constraint Group 14
Figure 5 Illustration of using constraint solving history 16
Figure 6 Performance w.r.t. different numbers of parameters 24
Figure 7 Performance w.r.t. different domain sizes 25
Figure 8 Performance w.r.t. different numbers of forbidden tuples 25
Figure 9 Comparison of Execution Time (2-way) 29
Figure 10 An example LTS graph and its exercised-after matrix 35
Figure 11 Real Example: JavaThread 39
Figure 12 Algorithm for t-way sequences generation 40
Figure 13 A target-oriented algorithm for test sequences generation 43
Figure 14 A brute force algorithm for test sequences generation 45
Figure 15 Illustration of test sequence extension 47
Figure 16 Illustration of test sequence extension 48
Figure 17 Example of SCC in an LTS graph 49
Figure 18 An algorithm for t-touring path generation 51
Figure 19 An SCC-base algorithm for test sequences generation 52
Figure 20 A Scenario of Using IEEE 11073 PHD Devices 63
Figure 21 Three Major Models in IEEE 11073 PHD 64
Figure 22 Manager State Machine (Flattened) 65
Figure 23 An Example Scenario of Data Exchange 67
Figure 24 An Overview of the Proposed Framework 68

Figure 25 The Architecture of Antidote 72

Figure 26 An Example of Transition Path (Manager) 74

Figure 27 Code Coverage Results 75

List of Tables

Table 1 Configurations of Real-Life Systems 18

Table 2 Configurations of Synthesized Systems 19

Table 3 Configurations of Optimization Options 20

Table 4 Comparison of Number of Constraint Solving Calls (2-way) 20

Table 5 Comparison of Execution Time (In Seconds) 21

Table 6 Test Generation with Different Test Strengths 23

Table 7 Comparison of Constrained Test Generation 27

Table 8 Comparison of Test Generation Algorithms 53

Table 9 Result of the Java Threads System 54

Table 10 Characteristics of Synthesized Systems 56

Table 11 Results of T-way Target Sequence Generation for Synthesized Systems 56

Table 12 Results of test sequence generation for synthesized systems (3-way) 57

Chapter 1

Introduction

Combinatorial testing (CT) has been shown to be a very effective testing strategy [1] [2] [3]. Given a system with n parameters, t-way combinatorial testing requires that all t-way combinations, i.e., all combinations involving any t parameter values, be covered by at least one test, where t is referred to as test strength and is typically a small number. A widely cited NIST study of several fault databases reports that all the faults in these databases are caused by no more than six factors [1]. If test parameters are modeled properly, t-way testing can expose all the faults involving no more than t parameters. This dissertation focuses on two important problems in combinatorial testing, including constrained test generation and combinatorial sequence testing.

First we study the problem of constrained test generation. Practical applications often have constraints on how parameter values can be combined in a test [4]. For example, one may want to ensure that a web application can be executed correctly in different web browsers running on different operating systems. Consider that Internet Explorer (or IE) 6.0 or later cannot be executed on MacOS. Thus, if the web browser is IE 6.0 or later, the operating system must not be MacOS. This constraint must be taken into account such that IE 6.0 or later and Mac OS do not appear in the same test.

The main challenge of constrained combinatorial test generation is dealing with this complexity, since both combinatorial testing and constraint solving are computation-intensive processes. We present an efficient algorithm, called IPOG-C, to address this challenge. Algorithm IPOG-C modifies an existing combinatorial test generation algorithm called IPOG [7] and employs a constraint solver to handle constraints. To optimize the performance of constraint handling, algorithm IPOG-C tries to reduce the number of calls

to the constraint solver. In case that such a call cannot be avoided, algorithm IPOG-C tries to simplify the solving process as much as possible.

We implemented algorithm IPOG-C in a combinatorial test generation tool called ACTS [8]. The experimental results on a set of real-life and synthesized systems indicate that the three optimizations employed in algorithm IPOG-C increased the performance by one or two orders of magnitude for most subject systems. Furthermore, the optimizations significantly slow down the increase in the number of calls to the constraint solver and the execution time as test strength, number of parameters, domain size, or number of forbidden tuples increases. Finally, a comparison of ACTS to three other tools suggests that ACTS can perform significantly better for systems with more complex constraints.

For the second problem, we study combinatorial sequence testing. Most work on combinatorial testing focuses on t-way test data generation, where each test is an (unordered) set of values for parameters. However, many programs exhibit sequence-related behaviors. Testing efforts for these programs should not only focus on data inputs, but also sequences of actions or events. The problem of t-way test sequence generation is fundamentally different from the problem of t-way test data generation in several aspects: (1) most t-way test data generation techniques assume that all the tests are of fixed length, which often equals the total number of parameters that are modeled. In contrast, test sequences are typically of various lengths, and this must be taken into account during t-way test sequence generation; (2) by the definition of “sequence”, t-way test sequence generation must deal with an extra dimension, i.e., “order”, which is insignificant in t-way test data generation; (3) sequencing constraints are different from, and typically more complex than, non-sequencing constraints. In particular, sequencing constraints need to be represented and checked in a way that is different from non-sequencing constraints.

In order to address the above challenges, we first introduce our system model, i.e., a labeled transition system, based on which we give a formal definition of t-way sequence coverage. This system model uses a graph structure to encode sequencing constraints. We divide the problem of t-way test sequence generation into two smaller problems, i.e., target sequence generation and test sequence generation. The first problem deals with how to generate the test requirements, i.e., all valid t-way sequences that must be covered. The second problem deals with how to generate a small set of test sequences to cover all the test requirements. We systematically explore different strategies to solve these problems and present a set of algorithms as the result of our exploration. We compare these algorithms both analytically and experimentally, with special attention paid to scalability. The experiment results show that while these algorithms have their own advantages and disadvantages, one of them is more scalable than others while exhibiting very good performance in test sequence generation.

The proposed t-way sequence coverage for a general system model can be used to model different types of programs, such as GUI applications, web applications, and concurrent programs. A set of algorithms for t-way test sequence generation is proposed, including an efficient algorithm for generating valid t-way sequences that must be covered, and four algorithms for generating a small set of test sequences that achieve the t-way sequence coverage. These algorithms are implemented in a Java application. We report an experimental evaluation of the proposed test generation algorithms that provides important insights about the advantages and disadvantages of these proposed algorithms. We point out t-way test sequence generation is the first stage of a larger effort that tries to expand the domain of t-way testing from test data generation to test sequence generation. Empirical studies on fault detection effectiveness of t-way test sequences are planned as the next stage of this larger effort.

Chapter 2

Constrained Test Generation

Practical applications often have constraints on how parameter values can be combined in a test [4]. For example, one may want to ensure that a web application can be executed correctly in different web browsers running on different operating systems. Consider that Internet Explorer (IE) cannot be executed on MacOS. Thus, if the web browser is IE, the operating system must not be MacOS. This constraint must be taken into account such that IE and Mac OS do not appear in the same test.

Constraints must be specified by the user before they are handled during test generation. One approach is to specify constraints as a set of forbidden tuples. A forbidden tuple is a value combination that should not appear in any test. When there are a large number of forbidden tuples, it can be difficult for the user to enumerate them. Alternatively, constraints can be specified as a set of logical expressions. A logical expression describes a condition that must be satisfied by every test. Logical expressions are more concise than explicit enumeration of forbidden tuples. In this paper, we assume that constraints are specified using logical expressions.

A major step in constraint handling is validity check, i.e., checking whether all the constraints are satisfied by a test. One approach to performing this check is to ensure that a test contains no forbidden tuples. This approach needs to maintain the complete list of all the forbidden tuples, which can be expensive when there are a large number of forbidden tuples. Alternatively, we can employ a constraint solver to perform this check. In this approach, we encode the problem of validity check as a constraint satisfaction problem. In this paper we focus on the latter approach, since it avoids maintaining the complete set of forbidden tuples and is thus a more scalable approach.

It is important to note that the way in which validity check is performed is independent from the way in which constraints are specified. For example, a tool called mAETG [5] uses forbidden tuples to specify constraints. Forbidden tuples are converted into a set of Boolean logic expressions, which are then solved by a SAT solver. In contrast, a tool called PICT [6] uses logic expressions to specify constraints. A list of forbidden tuples are first generated from the specified logic expressions and then used to perform validity check during test generation.

Both combinatorial testing and constraint solving are computation-intensive processes. The main challenge of constrained combinatorial test generation is dealing with this complexity. In this dissertation, we present an efficient algorithm, called IPOG-C, to address this challenge. Algorithm IPOG-C modifies an existing combinatorial test generation algorithm called IPOG [7] and employs a constraint solver to handle constraints. To optimize the performance of constraint handling, algorithm IPOG-C tries to reduce the number of calls to the constraint solver. In case that such a call cannot be avoided, algorithm IPOG-C tries to simplify the solving process as much as possible.

Specifically, algorithm IPOG-C includes the following three optimizations:

1) Avoiding unnecessary validity checks on t-way combinations. A t-way test set must cover all the valid t-way combinations. A t-way combination is valid if it can be covered by at least one valid test. Checking the validity of each t-way combination can be expensive since there often exist a large number of t-way combinations. The key insight in our optimization is that if a test is found valid, then all the combinations covered by this test would be valid, and thus do not have to be explicitly checked.

2) Checking relevant constraints only. When we perform a validity check, some constraints may not be relevant and thus do not have to be checked. We use a notion called constraint relation graph to identify groups of constraints that are related to

each other, which are then used to identify relevant constraints in a validity check. Algorithm IPOG builds a test set incrementally, i.e., covering one parameter at a time. This incremental framework is leveraged in this optimization to further reduce the number of relevant constraints that have to be involved in a validity check.

3) Recording the solving history. This optimization tries to reduce the number of calls to the constraint solver by saving previous solving results. This optimization works together with 2) to maximize reduction in the number of calls to the constraint solver.

For the purpose of evaluation, we implemented algorithm IPOG-C in a combinatorial test generation tool called ACTS. ACTS is freely available to the public [8]. We conducted experiments on a set of real-life and synthesized systems. The experimental results indicate that the three optimizations employed in algorithm IPOG-C increased the performance by one or two orders of magnitude for most subject systems. For example, for a real-life system GCC, the optimizations reduced the number of calls to the constraint solver from 34613 to 631 and the execution time from 683.599 seconds to 1.139 seconds. Furthermore, the optimizations significantly slow down the increase in the number of calls to the constraint solver and the execution time as test strength, number of parameters, domain size, or number of forbidden tuples increases. Finally, a comparison of ACTS to three other tools suggests that ACTS can perform significantly better for systems with more complex constraints.

2.1 Preliminaries

In this section, we formally define the problem of constrained combinatorial test generation.

Definition 1 (Parameter). A parameter p is a set of values, i.e., $p = \{v_1, v_2, \ldots, v_p\}$.

Value v for parameter p can be denoted as $p.v$. For ease of notation, we assume that different parameters are disjoint. This implies that each parameter value belongs to a unique parameter. This allows us to refer to a parameter value by itself, i.e., without mentioning which parameter it belongs to.

Definition 2 (Tuple). Let $G = \{p_1, p_2, \ldots, p_m\}$ be a set of parameters. A tuple $\tau = \{v_1, v_2, \ldots, v_m\}$ of G is a set of values where $v_i \in p_i$. That is, $\tau \in p_1 \times p_2 \ldots \times p_m$.

Intuitively, a tuple τ consists of a value v for each parameter p in a given set of parameters. We refer to a tuple of size t as a t-tuple. We also refer to v as the value of p in τ if there is no ambiguity. This effectively overloads the notion of a parameter, which may represent a set of values or may take a particular value, depending on its context. We use $\Pi(\{p_1, \ldots, p_m\})$ to denote $p_1 \times p_2 \ldots \times p_m$.

Definition 3 (SUT). A System Under Test (SUT) $M = <P, C>$ consists of a set $P = \{p_1, p_2, \ldots, p_{|P|}\}$ of parameters, where p_i is a parameter, and a set $C = \{c_1, c_2, \ldots, c_{|C|}\}$ of constraints, where each constraint c_i is a function: $\Pi(P) \rightarrow \{true, false\}$.

We refer to each tuple in $\Pi(P)$ as a test of M. In other words, a test is a special tuple whose size equals the number of parameters in a system. A constraint is a function that maps a test to a Boolean value *true* or *false*.

Definition 4. (Covering). A tuple τ is said to be covered by another tuple τ' if $\tau \subseteq \tau'$.

Note that a tuple is covered by itself. In this paper, we are particularly interested in the case where a tuple is covered by a test.

Definition 5. (Validity). Given a SUT $M = <P, C>$, a tuple τ of M is valid if $\exists\ \tau' \in \Pi(P)$, such that $\tau \subseteq \tau'$, and $\forall c \in C$, $c(\tau') = true$. Otherwise, τ is invalid.

If τ is a test, τ is valid if it satisfies all constraints. If τ is a t-tuple, where $t < |P|$, then τ is valid if there exists at least one valid test τ' that covers τ.

Definition 6. (Constrained T-Way Test Set). Let M = <*P*, *C*> be a SUT. Let Σ be the set of all valid t-tuples. A t-way constrained test set is a set $\Omega \subseteq \Pi(P)$ of tests such that, $\forall\ \sigma \in \Sigma$, there exists $\tau \in \Omega$ such that τ is valid and $\sigma \subseteq \tau$.

Intuitively, a constrained t-way test set is a set of valid tests in which each valid t-tuple is covered by at least one test. The problem of constrained t-way test generation is to generate a constrained t-way test set of minimal size. In practice, a tradeoff is often made between the size of the resulting test set and the time and space requirements.

2.2 The IPOG Algorithm

In this section, we introduce the original IPOG algorithm without constraint handling [5]. Due to space limit, we only present the major steps relevant to constraint handling. Refer to the original paper [5] for more details.

Algorithm IPOG works as follows: For a system with t or more parameters, we first build a t-way test set for the first t parameters. We then extend this test set to a t-way test set for the first (t+1) parameters, and continue to do so until it builds a t-way test set for all the parameters.

Assume that we already covered the first k parameters. To cover the (k+1)-th parameter, say p, it is sufficient to cover all the t-way combinations involving parameter p and any group of (t-1) parameters among the first k parameters. These combinations are covered in two steps, horizontal growth and vertical growth. Horizontal growth adds a value of p to each existing test. Each value is chosen such that it covers the most uncovered combinations. During vertical growth, the remaining combinations are covered one at a time, either by changing an existing test or by adding a new test. When we add a new test to cover a combination, parameters that are not involved in the combination are

given a special value called don't care. These don't care values can be later changed to cover other combinations.

Figure 1 illustrates how algorithm IPOG works. Assume that the system contains 4 parameters p1, p2, p3, and p4, and each parameter has 2 values {0, 1}. The test strength is 2. Assume that the 2-way test set for the first 3 parameters has been generated, as shown in Figure 1(a).

	p1	p2	p3
test 1	0	0	0
test 2	0	1	1
test 3	1	0	1
test 4	1	1	0

(a) The test set for p1, p2 and p3

p1	p4	p2	p4	p3	p4
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	1	0
1	1	1	1	1	1

(b) Target tuples for p4

	p1	p2	p3	p4
test 1	0	0	0	**0**
test 2	0	1	1	**1**
test 3	1	0	1	**0**
test 4	1	1	0	**0**

(c) Horizontal growth for p4

	p1	p2	p3	p4
test 1	0	0	0	0
test 2	0	1	1	1
test 3	1	0	1	0
test 4	1	1	0	0
test 5	**1**	**0**	**0**	**1**

(d) Vertical growth for p4

Figure 1 Illustration of the IPOG Algorithm

To cover the last parameter p4, we first generate all 2-way combinations that need to be covered. Figure 1(b) shows 12 2-way combinations to be covered. During horizontal growth, we add value 0 of P4 into the first test since it covers the most uncovered tuples {p1.0, p4.0}, {p2.0, p4.0} and {p3.0, p4.0}. Similarly, we add values 1, 0 and 0 of P4 into the next three tests, respectively, as shown in Figure 1(c). There are still 3 uncovered 2-way combinations, {p1.1, p4.1}, {p2.0, p4.1} and {p3.0, p4.1}. During vertical growth, we first generate a new test to cover {p1.1, p4.1}. Then we add p2.0 and

p3.0 into the same test to cover {p2.0, p4.1} and {p3.0, p4.1}, respectively. Figure 1(d) shows the complete 2-way test set.

2.3 The IPOG-C Algorithm

In this section, we modify algorithm IPOG to handle constraints. We refer to the new algorithm as IPOG-C. We first present a base version of algorithm IPOG-C. Then we propose three optimizations. The final version of algorithm IPOG-C is obtained by applying these optimizations to the base version. We also discuss how to apply these optimizations to other test generation algorithms.

2.3.1 The Base Version of Algorithm IPOG-C

Figure 2 shows the base version of the IPOG-C algorithm. The modifications made to the original IPOG algorithm are highlighted. These modifications do not change the main structure of the original IPOG algorithm. If no constraints are specified, the modified algorithm will generate the same test set as the original IPOG algorithm does.

Algorithm IPOG-C modifies the original IPOG algorithm to make sure: (1) all the valid t-way target tuples are covered; and (2) all the generated tests are valid. In line 5, we perform validity check on each t-way combination to identify all the valid t-way combinations that need to be covered. In lines 8 & 13, we perform the validity check to ensure that every test is valid. Since the algorithm terminates only when π is empty (line 12), all the valid t-way combinations must be covered upon termination.

```
Algorithm IPOG-C (int t, ParameterSet ps)
{
1. initialize test set ts to be an empty set
2. sort the parameters in set ps in a non-increasing order of their
       domain sizes, and denote them as P1, P2, …, and Pk
3. add into test set ts a test for each valid combination of values
       of the first t parameters
4. for (int i = t + 1; i ≤ k; i ++){
5.     let π be the set of all valid t-way combinations of values
           involving parameter Pi and any group of (t-1) parameters
           among the first i-1 parameters
6.     // horizontal growth for parameter Pi
7.     for (each test τ = (v1, v2, …, vi-1) in test set ts) {
8.         choose a value vi of Pi and replace τ with τ' = (v1, v2, …,
               vi-1, vi) so that τ' is valid and it covers the most
               number of combinations of values in π
9.         remove from π the combinations of values covered by τ'
10.    } // end for at line 7
11.    // vertical growth for parameter Pi
12.    for (each combination σ in set π){
13.        if (there exists a test τ in test set ts that can be changed to
               a valid test τ' that covers both τ and σ {
14.            change test τ to τ'
15.        } else {
16.            add a new test only contains σ to cover σ
17.        } // end if at line 13
18.    } // end for at line 12
19. }// end for at line 4
20. return ts;
}
```

Figure 2 The base version of the IPOG-C algorithm

2.3.2 Validity Check

Assume that we want to check the validity of a combination (or test) τ for a system *S*. This validity check problem is converted to a Constraint Satisfaction Problem (CSP), in which the variables are the parameters of *S*. The constraints include the constraints specified by the user and some constraints derived from τ, where each parameter value *p.v* in τ is represented by a constraint expression p = v. (Alternatively, one may change the parameter domain to a fixed value if it is supported by the solver.) A third party constraint solver is then used to solve this CSP.

Consider that a system consists of 3 parameters a, b, c, each having 3 values, and one constraint "a + b > c". Figure 3 shows the CSP for checking the validity of combination {a.0, b.0}. Note that two constraints "a = 0" and "b = 0" are added for parameter values a.0 and b.0, in addition to the user-specified constraint, i.e., "a + b > c".

[Variable]		[Constraints]	
a:	0, 1, 2	(1)	a + b > c
b:	0, 1, 2	(2)	a = 0
c:	0, 1, 2	(3)	b = 0

Figure 3 An example CSP problem

2.3.3 Optimizations

In this section, we propose several schemes to optimize the performance of constraint handling in algorithm IPOG-C.

A. Avoiding Unnecessary Validity Checks of Target Combinations

In line 5 of Figure 2, we first compute the complete set of all valid t-way combinations that need to be covered. This involves performing validity check on each t-way combination. This computation can be very expensive since there are typically a large number of t-way combinations.

We propose an optimization to reduce the number of validity checks on target combinations. The key observation is that there exists significant redundancy between validity checks for finding valid target tuples, and validity checks for choosing a valid parameter value during horizontal growth. That is, when we choose a new value, we perform validity check to ensure that the resulting test is valid. Since all the tuples covered in a test must be valid, this check implicitly checks validity of every tuple covered in this test. As a result, even though we do not have the list of all valid tuples, it is guaranteed that any tuple covered in a test, and removed from the target set π, must be valid.

The above observation suggests the following optimization. That is, we do not need to check the validity of target tuples (line 5 of Figure 2) before horizontal growth. It is important to note that the existence of invalid tuples in the target set (π) will not affect the greedy selection in horizontal growth (line 8 of Figure 2). This is because if a candidate test covers any invalid t-tuple, it must be an invalid test and will not be selected. So the effective comparison only happens between valid candidates.

After horizontal growth is finished, validity check needs to be performed on the remaining target combinations. That is, line 12 of Figure 2 should be changed to "for (each valid combination σ in set π)". At this point, many combinations are likely to have already been covered by horizontal growth. This means that the number of validity checks can be significantly reduced.

B. Checking Relevant Constraints Only

For a given validity check, some constraints may not be relevant, and thus do not need to be checked. In this optimization, we identify constraints that are relevant to a validity check and ignore the other ones. This helps to simplify the corresponding constraint solving problem.

We first divide constraints into non-intersecting groups. To do this, we use a graph structure called constraint relation graph, to represent relations between different constraints. In a constraint relation graph, each node represents a constraint, and each (undirected) edge indicates that two constraints have one or more common parameters. Then we find all the connected components in the graph. The constraints in each connected components are put into one group. Intuitively, constraints that share a common parameter, directly or indirectly, are grouped together.

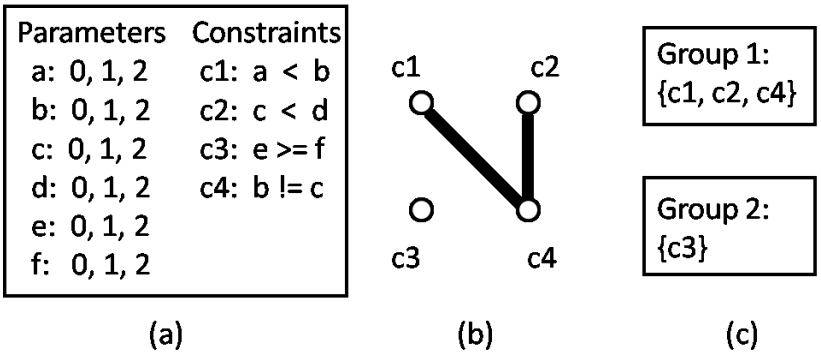


Figure 4 Illustration of Constraint Group

Figure 4(a) shows a system that contains 7 parameters and 4 constraints. Figure 4(b) shows the constraint relation graph for the system. There are two connected components. Figure 4(c) shows two constraint groups identified from the constraint relation graph. Note that this process only needs to be executed once. Similar techniques, which are often referred to as slicing, are used inside many constraint solvers.

Now we explain how to use constraint groups to identify irrelevant constraints. To check the validity of a test (or combination) τ, we identify relevant constraints as follows. For each parameter in τ, if it is involved in a constraint c, all the constraints in the same constraint group as c are identified to be relevant to this validity check. Only these constraints are encoded in the CSP sent to the constraint solver.

This optimization can be very effective considering that algorithm IPOG-C builds a test set incrementally. That is, we build a t-way test set for the first t parameters, and then extend this test set to cover the first t+1 parameters, and so on. When we try to cover a new parameter p, we only need to check the constraints in the same group as p.

Consider the example system in Figure 4(a). Assume we add a new parameter value f.0 to an existing test {a.0, b.1, c.0, d.0, e.1}, which must be valid. To check the validity of the new test, i.e., {a.0, b.1, c.0, d.0, e.1, f.0}, we only need to consider c3. Constraints c1, c2 and c4 are not relevant in this case.

C. Recording the Solving History

In this optimization, we record the solving history for each constraint group to avoid solving the same CSP multiple times. As discussed earlier, a CSP is encoded from a set of parameter values in the test that needs to be checked, and is then solved by a constraint solver. A Boolean value “true” or “false” that indicates the validity of the test will be returned. For each constraint solving call, we save solving history, i.e., the set of parameters values send to the CSP solver and the Boolean value returned by the CSP solver. Next time when a constraint solving call is going to make, we first search for the same set of parameters values in the solving history, and if a match is found, we can reused the cached result to avoid this solving call. Recall that in the previous optimization, we divide constraints into several non-intersecting groups. To increase the hit rate of the cached solving history, we divide a CSP problem into several independent sub-problems based on constraint groups, and then save the solving history for each of them.

Consider the example system in Figure 4(a). Assume that parameters a, b, c, d and e have been covered and we are trying to cover parameter f. Figure 5(a) shows 2 different candidate tests. As discussed earlier, the only relevant constraint is c3, which

involves parameters e and f. Therefore the validity of the two candidate tests is essentially determined by the combinations of values of parameters e and f in the two tests. Whereas the 2 tests that need to be checked are different, their validity is determined by the same value combination, i.e. {e.0, f.1}. Thus after checking the first test, we have the solving history for {e.0, f.1} (which is invalid), as shown in Figure 5(b). This allows us to derive that the second test is invalid without making a call to the constraint solver.

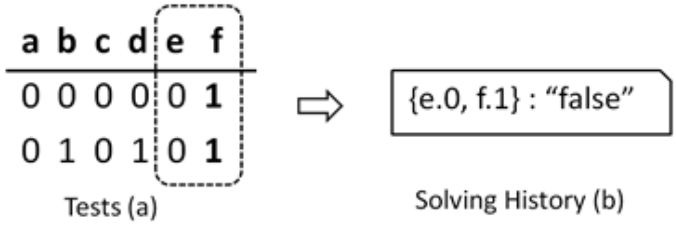


Figure 5 Illustration of using constraint solving history

2.3.4 Applying Optimizations to Other Algorithms

Our optimizations can be applied to other test generation algorithms. Due to space limitations, we discuss how to apply our optimizations to the AETG algorithm [12]. Like algorithm IPOG, the AETG algorithm adopts a greedy framework. However, it builds a test set one-test-at-a-time, instead of one-parameter-at-a-time.

The key to apply the first optimization is dealing with the existence of invalid t-way combinations in the target set, which is supposed to contain only valid t-way combinations. This could affect the greedy selection of a test value. In AETG, a test value is selected such that it covers the most valid combinations in the target set. According to the principle of the first optimization, all the combinations covered by a valid test are guaranteed to be valid. Thus, as long as the validity of the resulting test is checked after the selection of a test value, the existence of invalid combinations would not affect the

selection process. However, there is an exception with the selection of the first t - 1 values in a test. In the AETG algorithm, these values are selected such that they appear in the most number of combinations in the target set.

One approach to dealing with this exception is to change the AETG algorithm as follows. Instead of choosing the first t - 1 values one by one, we choose the first t values in a test altogether. This is done by finding the first valid t-way combination that remains in the target set, and then assign the t values in this combination to the test. This change makes the selection of the first t values less greedy. However, we note that t is typically small, and the selection of the other values remains unchanged.

The existence of invalid t-way combinations in the target set may also affect the termination condition. In the AETG algorithm, the test generation process terminates when the target set is empty. However, some invalid t-way combinations may never be covered, and thus the target set may never be empty. It is interesting to note that this problem can be resolved by the same change suggested earlier. That is, if we select the first t values of a test altogether by finding a valid t-way combination in the target set, the test generation process comes to a natural stop when no valid combination can be found.

The second and third optimizations do not interact with the core test generation process. That is, they take effect only during valid checks. Thus, they can be applied to the AETG algorithm without modifications.

2.4 Experiments

We implemented algorithm IPOG-C and integrated it into a combinatorial test generation tool called ACTS [6], which is freely available to the public. An open source CSP solver called Choco [13] is used for constraint solving.

Our experiments consist of three parts. The first part is designed to evaluate the effectiveness of the three optimizations. The second part is to investigate how the

performance of algorithm IPOG-C is affected by several factors, including test strength, number of parameters, size of domain and the number of forbidden tuples. The third part compares ACTS with other combinatorial test generation tools. All these experiments were performed on a laptop with Core i5 2410M 2.30GHz CPU and 4.0 GB memory, running 64-bit Windows 7.

2.4.1 Subject Systems

We use both real-life and synthesized systems in our experiments. The real-life systems include the five systems introduced in [14], and a system called TCAS introduced in [15]. We adopt the exponential notation in [14] to denote parameter configurations, where d^n means that there are n parameters with domain size d. The constraints in these systems were given in the form of forbidden tuples in [14] and [15]. We also use an exponential notation to denote constraints, where d^n means there are n forbidden tuples each of which involves d parameters. We manually convert each forbidden tuple to an equivalent constraint expression. For example, a forbidden tuple {a.0, b.1} is converted to a logic expression "! (a=0 && b=1)". The configurations of 6 real-life systems are listed in Table 1.

Table 1 Configurations of Real-Life Systems

Name	Num. of Parameters	Num. of Constraints	Parameter Configuration	Constraint Configuration
Apache	172	7	$2^{158}3^{8}4^{4}5^{1}6^{1}$	$2^{3}3^{1}4^{2}5^{1}$
Bugzilla	52	5	$2^{49}3^{1}4^{2}$	$2^{4}3^{1}$
GCC	199	40	$2^{189}3^{10}$	$2^{37}3^{3}$
SPIN-S	18	13	$2^{13}4^{5}$	2^{13}
SPIN-V	55	49	$2^{42}3^{2}4^{11}$	$2^{47}3^{2}$
TCAS	12	3	$2^{7}3^{2}4^{1}10^{2}$	2^{3}

We created 10 synthesized systems, all of which consist of 10 parameters of domain size 4. We denote the parameters as p1, p2, …, and p10. Each system contains a single constraint which is carefully designed to control the number of forbidden tuples. The number of forbidden tuples is an important measure of the complexity of a constraint. Note that the number of constraints is not important as different constraints can be joined together.

Some existing tools only support forbidden tuples as constraints. To compare to these tools, we derive all the forbidden tuples encoded by each constraint. Take system C1 as an example, we enumerate all 3-way value combinations of parameter p1, p2 and p3, and found 30 combinations that violate the constraint (p1>p2 || p3>p2) as forbidden tuples. We list the configurations, the number of derived forbidden tuples and detailed constraints for these synthesized systems in Table 2.

Table 2 Configurations of Synthesized Systems

Name	Param. Config.	Num. of Forbidd. Tuples	Constraint
C1	4^{10}	30	p1>p2 || p3>p2
C2	4^{10}	100	p1>p2 || p3>p4
C3	4^{10}	200	p1>p2 || p3>p4 || p5>p1
C4	4^{10}	300	p1>p2 || p3>p4 || p5>p2
C5	4^{10}	1000	p1>p2 || p3>p4 || p5>p6
C6	4^{10}	2000	p1>p2 || p3>p4 || p5>p6 || p7>p1
C7	4^{10}	3000	p1>p2 || p3>p4 || p5>p6 || p7>p2
C8	4^{10}	10000	p1>p2 || p3>p4 || p5>p6 || p7>p8
C9	4^{10}	20000	p1>p2 || p3>p4 || p5>p6 || p7>p8 || p9>p1
C10	4^{10}	30000	p1>p2 || p3>p4 || p5>p6 || p7>p8 || p9>p2

2.4.2 Evaluation of the Optimizations

To evaluate the effectiveness of individual optimizations and their combination, we tested multiple configurations. Table III shows five different configurations of the optimization options, where a tick denotes that the corresponding optimization is enabled, and dash means not. The first configuration represents the base version of IPOG-C, i.e., without any optimization, and the last one contains all the optimizations.

Table 3 Configurations of Optimization Options

Optimization	Base	O1	O2	O3	All
Avoiding unnecessary validity checks on t-way combinations	-	✓	-	-	✓
Checking relevant constraints only	-	-	✓	-	✓
Recording the solving history	-	-	-	✓	✓

Table 4 Comparison of Number of Constraint Solving Calls (2-way)

System	IPOG-C with Different Optimizations				
	Base	O1	O2	O3	All
Apache	15751	3903	12314	284	155
Bugzilla	2843	732	2352	57	50
GCC	34613	4753	31250	1032	631
SPIN-S	1183	478	1002	293	171
SPIN-V	10770	3679	9609	766	546
TCAS	828	597	535	59	42
C10	991	287	954	796	246

We use 6 real-life systems and the synthesized system with the most complex constraint, i.e., C10. The test strength is set to 2. We measure the performance of constrained test generation in terms of number of constraint solving calls (i.e., the

number of times the constraint solver is called) and execution time. The number of constraint solving calls is an importance metric because it is independent from the program implementation, the hardware configuration or different constraint solvers. The comparison results are shown in Table 4 and **Error! Not a valid bookmark self-reference.**. We do not show the number of tests, which is not affected by these optimizations.

Table 5 Comparison of Execution Time (In Seconds)

System	IPOG-C with Different Optimizations				
	Base	**O1**	**O2**	**O3**	**All**
Apache	105.411	6.225	9.403	0.687	0.577
Bugzilla	2.808	0.904	1.763	0.328	0.296
GCC	683.599	24.462	59.429	1.809	1.139
SPIN-S	1.545	0.92	1.31	0.749	0.53
SPIN-V	81.323	18.239	11.169	1.124	0.889
TCAS	0.874	0.749	0.702	0.36	0.328
C10	1.014	0.53	0.89	0.828	0.515

The results in Table 4 and We use 6 real-life systems and the synthesized system with the most complex constraint, i.e., C10. The test strength is set to 2. We measure the performance of constrained test generation in terms of number of constraint solving calls (i.e., the number of times the constraint solver is called) and execution time. The number of constraint solving calls is an importance metric because it is independent from the program implementation, the hardware configuration or different constraint solvers. The comparison results are shown in Table 4 and **Error! Not a valid bookmark**

self-reference.. We do not show the number of tests, which is not affected by these optimizations.

Table 5 suggest that the optimizations are very effective. Recall that the first optimization avoids validity check for all t-way target tuples in the beginning of test generation. This optimization is very effective when the system has a large number of t-way target tuples. For example, Apache contains 172 parameters and GCC has 199 parameters. They both have a large number of target tuples. With this optimization, the generation process runs 17 times faster for Apache, and 28 times faster for GCC.

The second optimization reduced the execution significantly, but not the number of constraint solving calls. This is because this optimization is aimed to simplify the actual constraint solving process by only considering relevant constraints. Note that the number of constraint solving classes is also slightly decreased. This is because a parameter may not be involved in any constraint. In this case, no relevant constraints are found and thus no constraints need to be solved.

The third optimization seems to be the most effective optimization in the experiments. Recall that it records the solving history based on constraint groups to reduce redundant solvings. This optimization is more effective with small constraint groups, where redundant solvings are more likely to happen. On the other hand, this optimization is less effective with large constraint groups. For example, for system C10, where 9 of 10 parameters belong to the same constraint group, this optimization is not very effective.

The three optimizations are complementary to each other and can be combined to further reduce the number of constraint solving calls and the execution time. In particular, for all of the real-life systems, the number of constraint solving calls was reduced by one or two orders of magnitude.

2.4.3 Evaluation of Different Factors

In this section, we explore how the performance of the entire test generation process is affected by different factors, including test strength, number of parameter, domain size and number of forbidden tuples. Each time we fix all the factors but one. We compare the test generation performance of our algorithm between with all the optimizations (i.e., the optimized version of algorithm IPOG-C) and without any optimization (the base version of algorithm IPOG-C).

A. Test Strength

We use system C1 to evaluate the performance of constrained test generation using different test strengths. Recall that C1 has 10 parameters of domain size 4. We record number of validity checks, number of times the constraint solver is called, and execution time in Table 6.

Table 6 Test Generation with Different Test Strengths

Test Strength	**Num. of Target Tuples**	**Base**		**Optimized**	
		Num. of Solving Calls	Time (sec)	Num. of Solving Calls	Time (sec)
2	683	644	0.67	77	0.31
3	7062	6869	3.9	121	0.32
4	47656	51787	64.17	122	0.40
5	218848	267421	1368.01	124	1.35
6	690816	Out of Memory	Out of Memory	124	14.39

One may find that as the test strength increases, the number of target tuples increase very fast. However, after those optimizations are applied, the number of constraint solving calls increases very slowly, and is even unchanged from strength 5 to 6. This is mainly due to the third optimization, which records the solving history for each

constraint group. This system contains a single constraint group involving only 3 parameters. After all of the possible $4^3 = 64$ value combinations, have been checked, all validity checks can be handled by looking up the solving history. That is, no more solving calls are needed.

B. Number of Parameters

In this section, we evaluate the performance of the test generation process with respect to different numbers of parameters. We built 8 systems with the number of parameters ranging from 4, 6, 8 to 18. The constraint "(p1>p2 || p3>p2)" in system C1 is used for all of these 8 systems. The test strength is set to 3.

Figure 6 show that as the number of parameters increases, the number of constraint solving calls and the execution time increase very fast for the base version, but very slow for the optimized version.

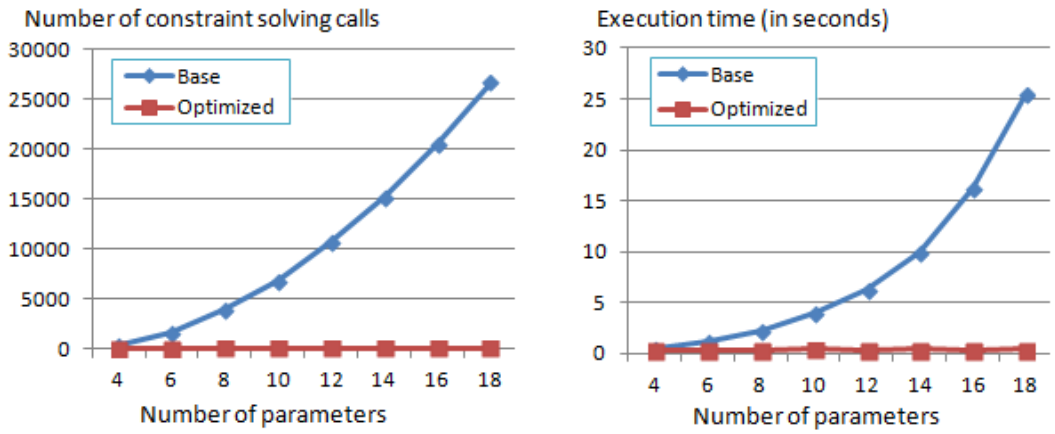


Figure 6 Performance w.r.t. different numbers of parameters

C. Domain Size

In this experiment, we still use system C1, but change the domain size to build 8 different systems. These systems have the same number of parameters and the same constraint, but the domain size increases from 2, 3, 4 until 9. The test strength is set to 3.

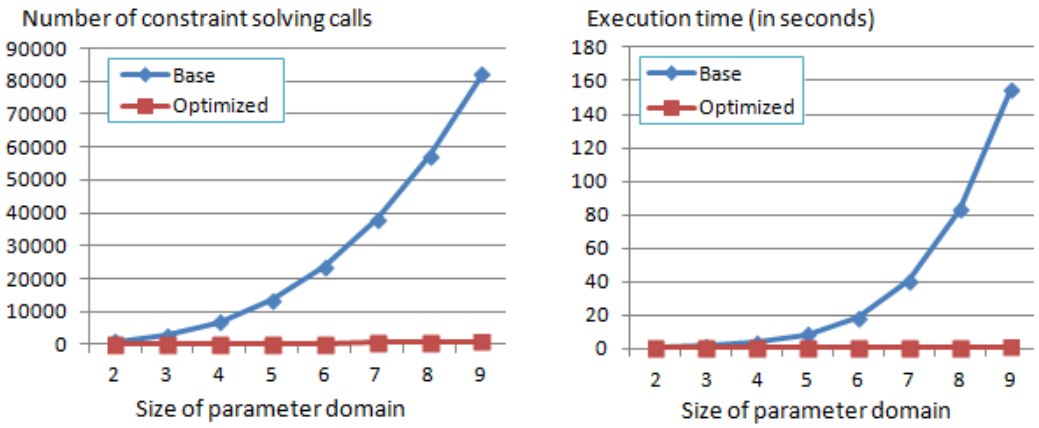


Figure 7 Performance w.r.t. different domain sizes

Again, Figure 7 shows that as the domain size increases, the number of constraint solving calls and the execution time increase very fast for the base version, but very slow for the optimized version.

D. Number of Forbidden Tuples

We use all of the 10 synthesized systems in this section to evaluate how the performance of the test generation process changes when the number of forbidden tuples changes. As discussed earlier, these systems have the same parameter configuration but different constraints. Those constraints are carefully designed to control the number of forbidden tuples. The test strength is set to 3.

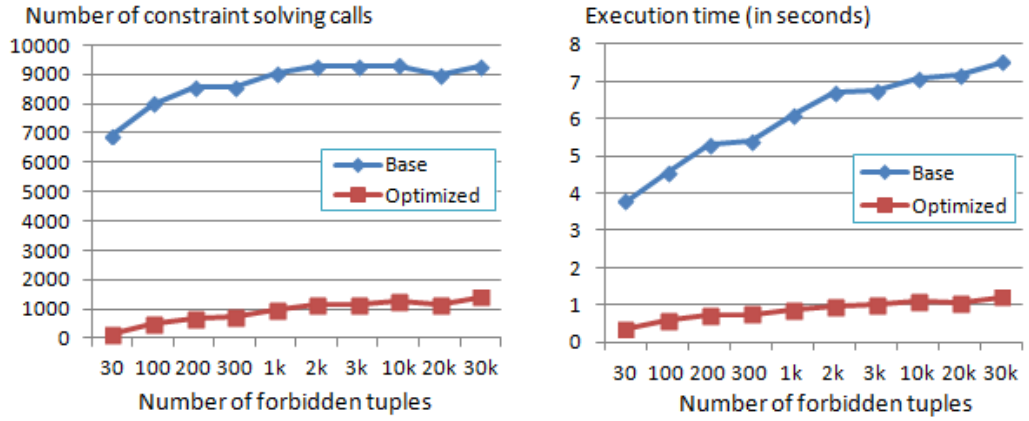


Figure 8 Performance w.r.t. different numbers of forbidden tuples

Figure 8 shows that as the number of forbidden tuples increases, the number of constraint solving calls and the execution time increase very fast for the base version, but very slow for the optimized version.

2.4.4 Comparison with Other Tools

In this section, we compare ACTS [6] (using the optimized IPOG-C algorithm) to other test generation tools. First we briefly introduce several existing test generation tools with constraint support.

CASA [14] integrates a SAT solver into a simulated annealing algorithm. Constraints are specified as Boolean formulas. We record the best result among five runs since this algorithm is not deterministic.

mAETG [10] integrates a SAT solver into an AETG-like algorithm. However, we did not make a comparison in this paper since mAETG is not available to public.

Ttuples [16] uses a greedy algorithm based on a property of (unconstrained) t-way test set, i.e., if two parameters have the same domain, it's safe to exchanging all their values. Constraints are specified as a set of forbidden tuples.

PICT [11] also adopts an AETG-like greedy algorithm. Constraints are specified in the form of logical expressions, but forbidden tuples are derived from the constraints to perform validity check.

These tools are compared in two dimensions: size of test set and execution time. However, it is important to note that size of test set mainly depends on the core test generation algorithms. Also even without constraints, the performances of different test generation algorithms are different. Furthermore, we did not make a comparison on the number of constraint solving calls or the number of validity checks, since we cannot obtain them from other tools.

Our comparison uses the 6 real-life systems and 10 synthesized systems introduced earlier. Since CASA and Ttuples cannot handle constraint expressions, we derived all forbidden tuples for 10 synthesized systems. The number of generated tests and the execution time for each system are shown in

Table 7. The number of forbidden tuples is also listed. The test strength is set to 3. We limit the execution time no more than 1000 seconds.

Table 7 Comparison of Constrained Test Generation

System	Num. of Forbid. Tuples	Ttuples		PICT		CASA		ACTS	
		size	time(s)	size	time(s)	size	time(s)	size	time(s)
Apache	7	-	>1000	202	176.01	-	>1000	173	25.2
Bugzilla	5	62	4.55	70	0.7	71	507.76	68	0.61
GCC	40	-	>1000	134	170.26	-	>1000	108	35.52
SPIN-S	13	127	0.3	113	0.09	103	187.51	98	1.82
SPIN-V	49	306	12.1	345	4.92	-	>1000	284	5.09
TCAS	3	402	0.27	409	0.11	405	99.7	405	0.55
C1	30	207	0.53	163	0.06	146	26.2	158	0.36
C2	100	202	1.31	171	0.06	164	47.23	168	0.58
C3	200	191	1.35	166	0.07			163	0.71
C4	300	200	2.69	166	0.08			161	0.74
C5	1000	196	5.03	170	0.18			160	0.87
C6	2000	195	7.12	163	0.96			161	0.97
C7	3000	188	14.94	162	1.09			160	1.00
C8	10000	196	257.75	163	17.34			164	1.11
C9	20000	-	>1000	162	242.1			157	1.06
C10	30000	-	>1000	161	461.5			158	1.21

We make several observations from Table 7. In the results for the 6 real-life systems, the execution time of CASA is much longer than others. This is because the simulated annealing algorithm is able to find a more optimal solution, but is usually much slower than greedy algorithms. Ttuples runs very fast, since it uses forbidden tuples instead of constraint solving to perform validity checks. PICT is also fast and generates good results. ACTS also generates relatively small test sets. ACTS runs slightly slower than Ttuples and PICT, but is still much faster than CASA. This is because the number of forbidden tuples for the real-life systems used in our experiments is very small. In this case, the strategy of using forbidden tuples is more efficient.

Compared to real-life systems, some synthesized systems have a large number of forbidden tuples. As the number of forbidden tuples increases, the execution time of Ttuples and PICT increases very fast. In contrast, the execution times of CASA and ACTS do not change much. This can be a significant advantage when we deal with systems with more complex constraints Note that the numbers of tests generated by these algorithms are very close.

Figure 9 shows how execution time changes as number of forbidden tuples increases. When the number of forbidden tuples is small, Ttuples, PICT and ACTS have similar execution time, while CASA takes much more time to finish. However, as the number of forbidden tuples increases, the execution time of Ttuples and PICT increases significantly. In contrast, the execution time of CASA increases slowly, and the execution time for ACTS remains almost unchanged. The reason is that CASA and ACTS use constraint solver for validity checks, and do not have to maintain a large number of forbidden tuples during test generation. This demonstrates a major advantage of using a constraint solver instead of forbidden tuples.

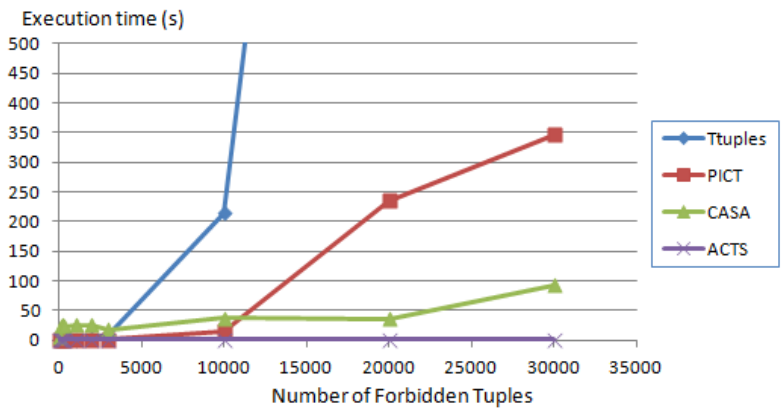


Figure 9 Comparison of Execution Time (2-way)

2.5 Related Work

We focus our discussion on work that handles constraints using a constraint solver. Garvin et al. integrated a SAT solver into a meta-heuristic search algorithm, called simulated annealing, for constrained combinatorial test generation [17] [18]. It was found that integration with the original version of the search algorithm did not produce competitive results, both in terms of number of tests and test generation time. Thus, a couple of changes were made to the original search algorithm to improve the results. The modified search algorithm could produce a different test set than the original search algorithm. This is in contrast to our work, where our optimizations do not change the original test generation algorithm, i.e., IPOG. In particular, our optimizations reduce the execution time spent on constraint handling, but the size of the test set will not be changed.

Cohen et al. integrated a SAT solver into an AETG-like test generation algorithm [10] [14]. They also proposed two optimizations to improve the overall performance. In their optimizations, the history of the SAT solver is exploited to reduce the search space of the original test generation algorithm. Like the work in [17], their optimizations require changes to the original test generation algorithm, and thus could produce a different test

set. In addition, their optimizations require access to the solving history and are thus tightly coupled with the SAT solver. This is in contrast with our optimizations, which do not change the original test generation algorithm and are independent from the constraint solver.

Recent work has applied combinatorial testing to software product lines. A software product line is a family of products that can be created by combining a set of common features. In this domain, constraint handling is a must because dependencies naturally exist between different features. Hervieu et al [19] developed a constraint programming approach for pairwise testing of software product lines. The focus of their work is on the conversion of the pairwise test generation problem to a constraint-programming problem. In particular, they formulated a global constraint to achieve pairwise coverage. Their work relies on the underlying constraint solver to achieve the best result. That is, they do not explicitly address the optimization problem.

Perrouin et al. [20] addressed the scalability of a constraint solver in the context of t-way testing of software product lines. Specifically, they address the problem that current constraint solvers have a limit in the number of clauses they can solve at once. They use a divide-and-conquer strategy to divide the t-way test generation problem for the entire feature model into several sub-problems. Their work addresses a different problem than, and is complementary to, our work, which tries to reduce the number of calls to a constraint solver and to remove constraints that are not relevant in a constraint solving call.

Johansen et al. [21] developed an algorithm called ICPL that applies t-way testing to software product lines. Similar to our algorithm, algorithm ICPL includes several optimizations to reduce the number of calls to a constraint solver. There are two major ideas in their optimizations that are closely related to our optimizations. The first idea is to

postpone removal of invalid target combinations (called t-sets in [21]). This achieves an effect similar to our first optimization, i.e., avoiding unnecessary validity checks of target combinations. However, there are two important differences. First, algorithm ICPL uses a heuristic to determine at which point to remove invalid target combinations. In contrast, our algorithm, IPOG-C, removes invalid target combinations during vertical growth, without using any heuristic condition. Second, they have very different motivations. Algorithm ICPL adopts a target combination-oriented framework, where the main loop iterates through the set of target combinations and covers them as they are encountered. Removing invalid combinations up front would cause two constraint solving calls for many valid combinations. (The other call is needed when a valid combination is actually covered in a test.) In contrast, our algorithm largely uses a test-oriented framework, where we try to determine each value in a test such that it covers as many combinations as possible. The key insight in our optimization is that if a test is found valid, then all the combinations covered by this test would be valid, and thus do not have to be explicitly checked.

The second optimization idea in algorithm ICPL that is closely related to ours is trying to check the validity of a t-way combination without actually calling the constraint solver. Algorithm ICPL is recursive in which a t-way test set is built by extending a (t-1)-way test set. The set of invalid combinations is maintained at each strength in the recursive process. An invalid t-way target combination is identified if it is an extension of an invalid (t-1)-way combination. In contrast, our algorithm records the solving history, which is used to determine the validity of a target combination as well as a test without calling the constraint solver. Also, our algorithm is not recursive, and does not maintain a set of invalid target combinations.

It is important to note that work on testing software product lines assumes Boolean parameters and constraints in the form of Boolean logic expressions. In contrast, our work does not have this restriction. Furthermore, software product lines typically have a large number of constraints but a small t-way test set. As a result, some optimizations that are effective for software product lines may not be very effective for general systems, and vice versa.

Finally we note that many optimization techniques are employed inside existing constraint solvers. In principle, our second and third optimizations are similar to constraint slicing and caching strategies used in some constraint solvers like zChaff [22] and STP [23]. These optimizations are also used outside a constraint solver in program analysis tools such as EXE [23]. However, we differ in that our optimizations work together with the combinatorial test generation algorithm and leverage its incremental framework to achieve maximal performance improvements. To our best knowledge, this is the first time these techniques are applied in a way that is integrated with the combinatorial test generation framework.

Chapter 3

Combinatorial Sequence Testing

Most work on combinatorial testing focuses on t-way test data generation, where each test is an (unordered) set of values for parameters. T-way combinatorial testing, or t-way testing, requires every combination of values for *any* t parameters be covered by at least one test. The rationale behind t-way testing is that many faults involve only a few parameters, thus testing all t-way combinations can effectively detect these faults. However, many programs exhibit sequence-related behaviors. For example, faults in graphical user interface (GUI) programs may only be triggered by a certain sequence of user actions [9]; faults in web applications may only be exposed when some pages are viewed in a certain order [10]; and faults in concurrent programs may not manifest unless some events are exercised in a particular order [11]. Testing efforts for these programs should not only focus on data inputs, but also sequences of actions or events.

In this dissertation, we study the problem of t-way test sequence generation. This problem is fundamentally different from the problem of t-way test data generation in several aspects: (1) Most t-way test data generation techniques assume that all the tests are of fixed length, which often equals the total number of parameters that are modeled. In contrast, test sequences are typically of various lengths, and this must be taken into account during t-way test sequence generation. (2) By the definition of “sequence”, t-way test sequence generation must deal with an extra dimension, i.e., “order”, which is insignificant in t-way test data generation. (3) Sequencing constraints are different from, and typically more complex than, non-sequencing constraints. In particular, sequencing constraints need to be represented and checked in a way that is different from non-sequencing constraints.

We first introduce our system model, *i.e.*, a labeled transition system, based on which we give a formal definition of t-way sequence coverage. This system model uses a graph structure to encode sequencing constraints. We divide the problem of t-way test sequence generation into two smaller problems, i.e., target sequence generation and test sequence generation. The first problem deals with how to generate the test requirements, i.e., all valid t-way sequences that must be covered. The second problem deals with how to generate a small set of test sequences to cover all the test requirements. We systematically explore different strategies to solve these problems and present a set of algorithms as the result of our exploration. We compare these algorithms both analytically and experimentally, with special attention paid to scalability. The experiment results show that while these algorithms have their own advantages and disadvantages, one of them is more scalable than others while exhibiting very good performance in test sequence generation.

We point out that we currently focuses on t-way test sequence generation, which is the first stage of a larger effort that tries to expand the domain of t-way testing from test data generation to test sequence generation. Empirical studies on fault detection effectiveness of t-way test sequences are planned as the next stage of this larger effort.

3.1 Preliminaries

3.1.1 System Model

We model a system under test as a labeled transition system (LTS).

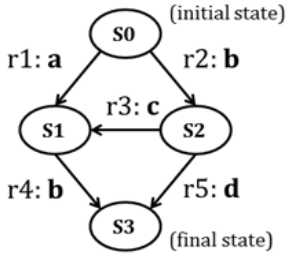
Definition 1. A labeled transition system M is a tuple $<S, S_I, S_f, L, R>$, where S is a set of states, $S_I \subseteq S$ is a set of initial states, $S_f \subseteq S$ is a set of final states, L is a set of event labels, and $R \subseteq S \times L \times S$ is a set of labeled transitions.

An LTS can be built from system requirements, high-/low-level designs, or implementations at a certain level of abstraction. The size of an LTS can be controlled by

choosing an appropriate level of abstraction and by modeling system parts that are of interest, instead of the whole system. Initial states in an LTS represent states where an execution of system could start, and final states represent states where a system could safely stop.

For ease of reference, we will refer to a labeled transition as a transition, and refer to an event label as an event. For a transition r, we use $src(r) \in S$ to denote the source state of r, $dest(r) \in S$ to denote the destination state of r, and $event(r) \in L$ to denote the event label of r.

An LTS can be represented by a directed graph, in which each vertex represents a state, and each directed edge represents a transition between two states and is labeled with an event. An example graph is shown in Figure 10(a). We refer to such a graph as an LTS graph.



	r1	r2	r3	r4	r5
r1	0	0	0	1	0
r2	0	0	1	1	1
r3	0	0	0	1	0
r4	0	0	0	0	0
r5	0	0	0	0	0

(b)

Figure 10 An example LTS graph and its exercised-after matrix

Definition 2. A transition sequence $p = r_1 \bullet r_2 \bullet \ldots \bullet r_n$ is a sequence of n transitions $< r_1, r_2, \ldots, r_n >$ such that $dest(r_i) = src(r_{i+1})$, for $i=1,2, \ldots, n-1$.

Intuitively, a transition sequence represents a path in an LTS graph. We use these two terms interchangeably. Given a transition sequence $P = r_1 \bullet r_2 \bullet \ldots \bullet r_n$, we denote the corresponding event sequence $event(P) = event(r_1) \bullet event(r_2) \bullet \ldots \bullet event(r_n)$. We use $event(P)$ to represent P if there is no ambiguity, i.e., only one test sequence can

be represented by *event*(*P*). We distinguish the notion of *transition sequence* from the notion of *sequence of transitions*. The former requires transitions to be consecutive in a sequence, whereas the latter does not require so. In the rest of this paper, we use $r_1 \bullet r_2 \bullet \ldots \bullet r_n$ to indicate a transition sequence, and use $< r_1, r_2, \ldots, r_n >$ to indicate a sequence of transitions.

Definition 3. Let *r* and *s* be two transitions in a labeled transition system *M*. Transition *s* can be *exercised after r*, denoted as $r \rightarrow s$, if there exists a transition sequence $P = r_1 \bullet r_2 \bullet \ldots \bullet r_n$, where $n > 0$, such that $r_1 = r$ and $r_n = s$.

As a special case, if $r \bullet s$ is a transition sequence, i.e., *dest*(*r*) = *src*(*s*), it is said that transition *s* can be exercised *immediately* after *r*. If transition *s* can be *exercised after* transition *r*, it is also said that transition *r* can be *exercised before* transition *s.* Consider the example system in Figure 1(a). Transition r_2 can be exercised before r_3, r_4 and r_5. A transition may be exercised after (or before) itself, implying the existence of a cycle in the LTS graph.

Definition 4. Let *e* and *f* be two events in an LTS *M*. Event *f* can be exercised after event *e*, denoted as $e \rightarrow f$, if there exist two transitions *r* and *s* such that *event*(*r*)=*e*, *event*(*s*)=*f*, and $r \rightarrow s$.

While the exercised-after relation between transitions is transitive, the exercised-after relation between events is not. For example, in Figure 1(a), for events *a*, *b* and *d*, we have $a \rightarrow b$, $b \rightarrow d$, but $a \rightarrow d$ is inconsistent with the LTS graph.

Assume that there are *n* transitions in a labeled transition system. We can build a $n \times n$ matrix *E*, where *E*(*i,j*)=1 if $r_i \rightarrow r_j$, and *E*(*i,j*)=0 otherwise. This exercised-after matrix can be constructed in $O(n^3)$ time, using Warshall's algorithm [25]. Figure 10 shows the exercised-after matrix for the LTS graph.

3.1.2 T-way Sequence Coverage

We define the notion of t-way sequence coverage in terms of t-way target sequences and test sequences. T-way target sequences are test requirements, i.e., sequences that must be covered; test sequences are test cases that are generated to cover all the t-way target sequences.

Definition 5. A t-way target sequence is a sequence of *t* events $<e_1, e_2, ..., e_t>$ such that there exists a single sequence of *t* transitions $<r_1, r_2, \ldots, r_t>$ where $r_i \rightarrow r_{i+1}$, and $e_i = label(r_i)$ for $i=1,2, \ldots, t$.

Intuitively, a t-way target sequence is a sequence of *t* events that could be exercised in the given order, consecutively or inconsecutively, by a single system execution. The same event could be exercised for multiple times in a t-way target sequence. Note that not every sequence of *t* events is a t-way target sequence. For example, in Figure 1(a), <*a, c*>, <*a, d*>, and <*c, d*> are not 2-way target sequences.

One may attempt to define a t-way target sequence as a sequence of *t* events $<e_1, e_2, ..., e_t>$ in which $e_i \rightarrow e_{i+1}$, for $i=1,2, \ldots, t$. This definition is however incorrect. For example, consider a sequence of 3 events <*a, b, c*> in Figure 1(a). We have a→b and b→c. However, <*a, b, c*> cannot be executed in a single transition sequence, and therefore is not a t-way target sequence.

Definition 6. A test sequence is a transition sequence $r_1 \bullet r_2 \bullet \ldots \bullet r_n$, where $src(r_1) \in S_i$, $dest(r_n) \in S_f$.

Definition 7. A test sequence $P = r_1 \bullet r_2 \bullet \ldots \bullet r_n$ covers a t-way target sequence $Q=<e_1, e_2, ..., e_t>$, if there exist $1 \le i_1 < i_2 < ... < i_t \le n$ such that $event(r_{i_k}) = e_k$, where $k = 1,2,\ldots, t$.

A test sequence is a transition sequence that starts from an initial state and ends with a final state in the LTS graph. Intuitively, a test sequence is a (complete) transition sequence that can be exercised by a test execution. A target sequence *Q* is covered by a test sequence *P* if all the events in *Q* appear in *event*(*P)* in order. For example, in Figure 1(a), a test sequence $r_2 \bullet r_3 \bullet r_4$ covers three 2-way target sequences: <*b, c*>, <*c, b*>, and <*b, b*>.

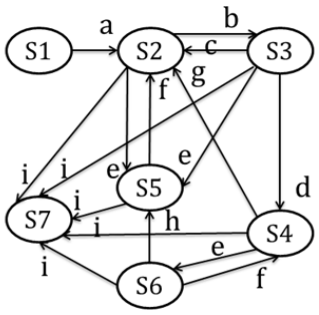
Definition 8. Given an LTS *M*, let Σ be the set of all t-way target sequences. A t-way test sequence set Π is a set of test sequences such that for $\forall Q \in \Sigma$, $\exists P \in \Pi$ that *P* covers *Q*. Integer *t* is referred as the *test strength*.

The t-way sequence coverage requires that every t-way target sequence be covered by at least one test sequence. Consider the example in Figure 10(a). Three test sequences, $r_1 \bullet r_4$, $r_2 \bullet r_5$ and $r_2 \bullet r_3 \bullet r_4$ covers all 2-way target sequences <*b, b*>, <*b, c*>, <*b, d*>, <*c, b*> and <*a, b*>.

The notion of t-way sequence coverage is similar to *all-transition-k-tuples* coverage for web applications [24] and *length-n-event-sequence* coverage for GUI applications [21]. All these coverage criteria require sequences of a certain number of events be covered. However, they differ in that t-way sequence coverage does not require events in a target sequence to be covered consecutively by a test sequence whereas the other two require so. This difference has a significant implication on test sequence generation, which is illustrated below.

Figure 11 shows a labeled transition system that represents the life cycle of Java threads. This system is later used as a subject system in our experiments in Section V.A. Assume that a fault can be exposed only if a sequence of three events, “start”, “IO completes”, “notify” (or event sequence <*a, h, g*>), is exercised. A shortest path (transition sequence) that covers <*a, h, g*>is *a•b•d•e•h•f•b•d•g*, which is of length 9. If

length-n-event-sequence coverage is used, *all* paths of length 9 must be covered in order to cover this faulty sequence <*a, h, g*>. This can significantly increase the number of test sequences. In contrast, 3-way sequence testing may generate a transition sequence of length 9 for this target sequence, but it does not require all paths of length 9 be covered. This significantly reduces the number of test sequences while still detecting this fault.



	States
S1	NEW (initial)
S2	RUNNABLE
S3	RUNNING
S4	BLOCKED
S5	SUSPENDED
S6	SUSPEND BLOCKED
S7	DEAD (final)

	Events
a	start
b	scheduled
c	yield
d	sleep, wait, join..
e	suspend
f	resume
g	notify, notifyAll...
h	IO completes
i	stop, run ends

Figure 11 Real Example: JavaThread

3.2 Target Sequence Generation

Assume that a system contains *n* events. There are at most n^t t-way target sequences. Some t-way sequences, i.e., sequences of t events, are not valid target sequences due to constraints encoded by the transition structure. One approach is to first generate all possible t-way sequences and then filter out those sequences that are not valid target sequences. A t-way sequence is a valid target sequence if there exists a transition sequence that covers this sequence. This approach is however not efficient. In the following we describe a more efficient, incremental approach to generate target sequences.

The main idea of our approach is to first generate all 2-way target sequences, and then extend them to generate all 3-way target sequences, and continue to do so until we generate all the t-way target sequences. Given an LTS *M*, we first build the exercised-after matrix for all the transitions of *M*. Next we find the set of all possible transition pairs

<*r, r'*> that *r'* can be exercised after *r*. We refer to this set as the 2-way transition sequence set. For each 2-way transition sequence <*r, r'*>, and for each transition *r''* that can be exercised after *r'*, we generate a 3-way transition sequence <*r, r', r''*>. These 3-way transition sequences constitute the 3-way transition sequence set. We repeat this process until we build the *t*-way transition sequence set. At this point, we convert each t-way transition sequence to a t-way target sequence. Figure 12 shows the detailed algorithm that implements this approach.

Algorithm: **GenTargetSeqs**
Input: a LTS *M*, test strength *t*
Output: a set Σ consisting of all the *t*-way target sequences
1. Let $r_1, r_2, \ldots, r_n$ be all the transitions in *M*
2. build the exercised-after matrix *E* such that *E(i,j)*=true if $r_i \rightarrow r_j$, and *E(i,j)*=false otherwise
3. let Ω_i, 1 <= *i* <= *t*, be an empty set
4. add into Ω_1 each transition $r_1, r_2, \ldots, r_n$ as a single-transition sequence
5. for (*k* = 2 to *t*){
6. for (each sequence of transitions *R* in Ω_{k-1}) {
7. let r_j be the last transition of *R*
8. for(*i* = 1 to *n*) {
9. if (*E*(*j, i*) == true)
10. add sequence of transitions <*R*, r_i> to Ω_k
11. }
12. }
13. }
14. for (each sequence of transitions *R* = <$r_{j1}, r_{j2}, \ldots, r_{jt}$>in Ω_t){
15. add into Σ an event sequence <*event*(r_{j1}), *event*(r_{j2}), … *event*(r_{jt})> if not exists
16. }
17. return Σ

Figure 12 Algorithm for t-way sequences generation

Complexity analysis: Assume there are n transitions, and the test strength is t. There are at most n^t t-way transition sequences. As we discussed earlier, the time complexity for line 2 is $O(n^3)$, The time complexity for line 5 to 13 is dominated by the last iteration, which is $O(n^t)$. Line 14 to 16 takes $O(tn^t)$ time. The time complexity of the entire algorithm is $O(n^3+tn^t)$, which is $O(n^3)$ when t<3, and $O(tn^t)$ otherwise. The space complexity is $O(n^2+tn^t)$ since we store the exercised-after matrix and all t-way transition sequences.

In the next section we will present four test sequence generation algorithms based on t-way target sequences generated by *GenTargetSeqs*.

3.3 Test Sequence Generation

The objective of test sequence generation is to generate a set of test sequences that covers all t-way target sequences and that requires minimal test effort. There are different factors to be considered. The more transitions exist in a test sequence, the longer time it takes to execute. Thus, it is typically desired to minimize the total length of these test sequences. Also, we often need to set up the environment before we execute a test sequence, and tear down the environment after we finish. Thus, in order to minimize such setup and teardown costs, it is often desired to reduce the number of test sequences.

In this section, we present four algorithms for test sequence generation. While these algorithms are mainly designed to minimize the total length of test sequences, they can be modified to support other types of cost optimization. These algorithms are resulted from a systematic exploration of possible strategies for test sequence generation.

3.3.1 A Target-Oriented Algorithm

Recall that in algorithm *GenTargetSeqs*, each t-way target sequence is derived from a sequence of *t* transitions. The main idea of this test generation algorithm is that, for each target sequence *Q*, we find a shortest test sequence to cover *Q* by extending the sequence of transitions from which *Q* is derived. Since a test sequence typically covers many target sequences, a greedy algorithm is then used to select a small subset of these test sequences that also covers all the target sequences.

The key challenge in this algorithm is how to generate a shortest test sequence to cover each target sequence. Let *Q* be a target sequence and *C* be the sequence of transitions from which *Q* is derived. We first extend *C* to a transition sequence *P* by inserting a transition sequence between every two adjacent transitions that are not consecutive. For example, assume $C = \langle r_1, r_2, \ldots, r_t \rangle$. If r_{i+1} cannot be exercised immediately after r_i, we insert a shortest transition sequence between $dest(r_i)$ and $src(r_{i+1})$. This path always exists, as otherwise *Q* cannot be a t-way target sequence. Next, we make *P* a test sequence by inserting a shortest path from an initial state to the beginning state of *P* and a shortest path from the ending state of *P* to a final state, if necessary. The test sequence *P*, as constructed so, is a shortest test sequence that covers *Q*. Figure 13 shows the details of this algorithm.

Note that in line 2, all shortest paths between two nodes in a graph can be effectively calculated using Floyd-Warshall algorithm [25]. During target sequence generation, we keep all the sequences of transitions from which the target sequences are derived. These sequences are used in line 5. A greedy algorithm is used in line 17 to select a subset of test sequences that can cover all the target sequences. This algorithm is similar to a classic greedy set cover algorithm [25]. It works iteratively, i.e., at each

iteration we choose a test sequence that covers the most number of uncovered target sequences, and remove covered ones, until all target sequences are covered.

Algorithm: **GenTestSeqsFromTargets**
Input: an LTS M, a set Σ of target sequences, test strength t
Output: a t-way test sequence set Π
1. let Π and Ω be an empty set of test sequences
2. find a shortest path for every pair of states
3. for each target sequence $Q = <e_1, e_2, \ldots, e_t>$ in Σ {
4. let P be an empty transition sequence
5. let $C = <r_1, r_2, \ldots, r_t>$ be a sequence of transitions such that Q is derived from C
6. for (i = 1 to t - 1){
7. append to P a shortest path from $dest(r_i)$ to $src(r_{i+1})$
8. }
9. if ($src(r_1)$ is not an initial state) {
10. append to P a shortest path from an initial state to $src(r_1)$
11. }
12. if ($dest(r_t)$ is not a final state){
13. append to P a shortest path from $dest(r_i)$ to a final state
14. }
15. add test sequence P into Ω.
16. }
17. use a greedy algorithm to select a subset Π of Ω that can cover all the target sequences in Σ
18. return Π

Figure 13 A target-oriented algorithm for test sequences generation

Complexity analysis: Assume there are n transitions and $|\Sigma|$ target sequence in an LTS. Assume the test strength is t, and m test sequences are generated. Finding all pair-wise shortest paths takes $O(n^3)$ time [25]. In order to build a shortest test sequence for each target sequence, we have to append at most (t+1) shortest paths, which takes

O(t) time. Thus the total time from line 3 to 16 is O($t|\Sigma|$). The greedy algorithm in line 17 takes O($m|\Sigma|^2$) time. Therefore the time complexity for the entire algorithm is O($n^3+t|\Sigma|+m|\Sigma|^2$) = O($n^3+ m|\Sigma|^2$). The space complexity is O($n^2+|\Sigma|$).

3.3.2 A Brute Force Algorithm

This algorithm finds all test sequences of length up to h, in a brute force manner. Then it selects a subset of these test sequences to cover as many target sequences as possible. Note that the value of h is specified by the user. This algorithm is used as a baseline in our effort to develop more efficient algorithms.

The first step of this algorithm uses a strategy similar to the breadth-first search (BFS) to generate all test sequences of length up to h. The difference is that in BFS, a node is only explored once, while in our strategy, a node is explored multiple times as it may lead to different test sequences. After all test sequences of length up to h are found, we apply the same greedy algorithm as mentioned early to select a subset of test sequences to cover as many target sequences as possible. If there are any target sequences that remain uncovered, we use algorithm *GenTestSeqsFromTargets* to cover them. Figure 14 shows the detailed algorithm, named *GenTestSeqsBF*.

For each target sequence, we can generate a shortest test sequence to cover it. This information can be used to select a proper value for h. In particular, if we set h to the maximal length of all these shortest test sequences, then all the target sequences are guaranteed to be covered, because all test sequences of length up to h will be found in this algorithm. However the number of test sequences of length up to h increases exponentially with respect to h, so do the execution time and memory cost.

Algorithm: **GenTestSeqsBF**

Input: a LTS M, a target sequence set Σ, test strength t, max test sequence length h

Output: a t-way test sequence set Π

1. use a BFS-like algorithm to generate the set Ω of all the test sequences of length up to h
2. use a greedy algorithm to select a subset $\Pi \subseteq \Omega$ that covers all the target sequences in Σ that could be covered.
3. remove from Σ the target sequences covered by Π
4. if (Σ is not empty) {
5. use algorithm *GenTestSeqsFromTargets* to generate a test sequence set Π' to cover the target sequences in Σ
6. $\Pi = \Pi \cup \Pi'$
7. }
8. return Π

Figure 14 A brute force algorithm for test sequences generation

Complexity analysis: Assume there are n transitions, and $|\Sigma|$ target sequences in an LTS. Assume the test strength is t, maximal length is h, and m test sequences are generated. There are at most n^h candidates of length up to h, which takes $O(n^h)$ to generate. The greedy algorithm in line 2 takes $O(|\Sigma|mn^h)$ time, as discussed earlier. The entire algorithm is dominated by line 2, and the total time complexity is $O(|\Sigma|mn^h)$. We have to store all n^h candidate test sequences, so the total space complexity is $O(|\Sigma|+n^h)$.

3.3.3 An Incremental Extension Algorithm

This new algorithm is motivated by the observation that, a longer test sequence covers more t-way targets. However, when the length of test sequence increases, the number of all possible test sequences increases exponentially. Thus, it is not practical to generate long test sequences using algorithm *GenTestSeqsBF*. This new algorithm is designed to generate longer test sequences by adopting an incremental framework. In

this framework, a test sequence is generated in multiple steps. At each step, we generate all test sequences of a given length, which is often small, and then select one of these sequences such that appending this sequence to the sequence obtained from the previous steps will cover the most target sequences. Once a complete test sequence is generated, either because we reach a final state or can no longer make progress, we remove the target sequences covered by this sequence and then repeat the same process to generate the next test sequence. We keep doing so until no more targets can be covered. The remaining targets, if exist, are covered by algorithm *GenTestSeqsFromTarget*.

The incremental extension is performed as follows. Given a transition sequence $P = r_1 \bullet r_2 \bullet \ldots \bullet r_n$, we generate the set *Q* of all *maximal* transition sequences that start from the ending state of this sequence, i.e., $dest(r_n)$, and that are of length up to *h*. A maximal transition sequence of length up to *h* is a transition sequence that is not a prefix of any other transition sequence of length up to *h.* A transition sequence *P'* is selected from *Q* such that sequence $P \bullet P'$ covers the most number of target sequences. Then we set $P = P \bullet P'$, *i.e.,* appending *P'* to *P*. An example of incremental extension is illustrated in Figure 15. In the previous extension (step *k-1*), a transition sequence that ends with transition *a* being the last transition was found. Now we explore all maximal transition sequences staring from *dest*(*a*) and of length up to 3. A transition sequence $b \bullet c \bullet d$ is selected as it covers the most number of target sequences. This process is then repeated and the next extension will start from *dest*(*d*). This extension is terminated when it reaches a final transition (successfully generated a long test sequence), or no targets can be covered (terminated).

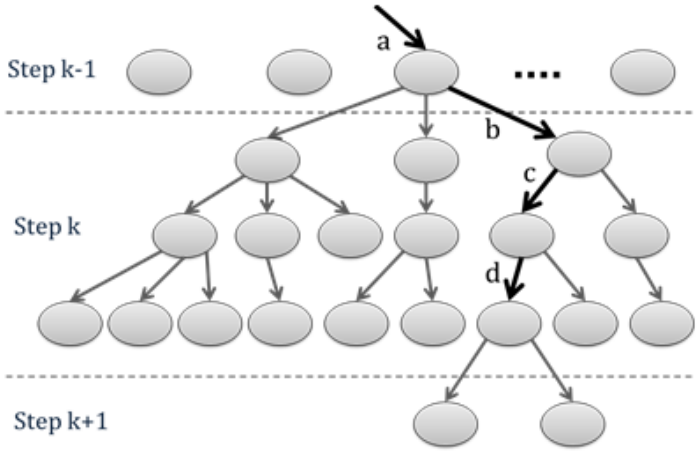


Figure 15 Illustration of test sequence extension

Figure 16 shows the detailed *GenTestSeqsInc* algorithm. The parameter $h>t$ indicates the search depth in each extension. Due to the incremental nature, the value of h in algorithm *GenTestSeqsInc* can be much smaller than in algorithm *GenTestSeqsBF*. Note that the extension process may get stuck in a long cycle, thus making no progress (line 9). In this case, the extension process is terminated and the *GenTestSeqsFromTargets* algorithm is used to cover any remaining targets.

Complexity analysis: Assume that there are n transitions and $|\Sigma|$ t-way target sequences in an LTS. Also assume that the test strength is t, the search depth for each extension is h. Assume that there is a total of d extensions. For each extension, there are at most n^h possible transition sequence of length up to h, and it takes $O(|\Sigma|n^h)$ to find the best one. Therefore the time complexity for the entire algorithm is $O(|\Sigma|dn^h)$. The space complexity is $O(|\Sigma|+n^h)$, since we only have to keep the possible transition sequences for one extension at a time.

Algorithm: **GenTestSeqsInc**

Input: an LTS *M*, a set *Σ* of target sequences, test strength *t*, search depth *h*

Output: a *t*-way test sequence set *Π*

1. initialize *Π* to be empty
2. let *S* be a set consisting of all the initial states
3. let *P* be an empty test sequence
4. while (true){
5. use a BFS-like algorithm to generate the set *Ω* of all maximal transition sequences that begins with a state in *S* and that are of length up to *h*
6. select a transition sequence *P'* in *Ω* such that the extended transition sequence *P* • *P'* covers the most targets
7. let *P* = *P* • *P'*, and *s* be the last transition of *P*
8. clear *S* and add *s* to *S*
9. if (*P* covers no target sequence) {
10. break //may be stuck in a cycle
11. }else if (*s* is a final state) {
12. add *P* to *Π*
13. remove target sequences covered by *P* from *Σ*
14. reset *S* to a set consisting of all the initial states
15. reset *P* to an empty test sequence
16. }
17. }
18. if (*Σ* is not empty) {
19. use algorithm *GenTestSeqsFromTargets* to generate a test sequence set *Π'* to cover remaining targets
20. *Π* = *Π* ∪ *Π'*
21. }
22. return *Π*

Figure 16 Illustration of test sequence extension

3.3.4 An SCC-Base Algorithm

One of the biggest challenges in test sequence generation is how to handle transition cycles, as they could cause transition sequences to be extended infinitely. To address this problem, algorithm *GenTestSeqsBF* limits the maximal length of each test sequence, and algorithm *GenTestSeqsInc* terminates the extension process when no progress is made due to a cycle. We use a different strategy to treat cycles in this SCC-based algorithm, which has three major steps. In the first step, we build an acyclic LTS *M'* from the original LTS *M.* In the second step, we generate test sequences for the acyclic LTS *M'*, referred as *abstract paths* as they need to be mapped back to the original LTS later. In the last step, we extend all the abstract paths to test sequences for the original LTS *M*, such that all target sequences are covered. We will explain each step with more details.

A. Build Acyclic LTS

In graph theory, a strongly connected component (SCC) is defined as a graph in which every two nodes can reach each other. An SCC detection algorithm can be found in [25]. We can collapse each SCC in a LTS graph to a special node, called an SCC node. Doing this converts the original LTS to an acyclic LTS. An example is shown in Figure 17.

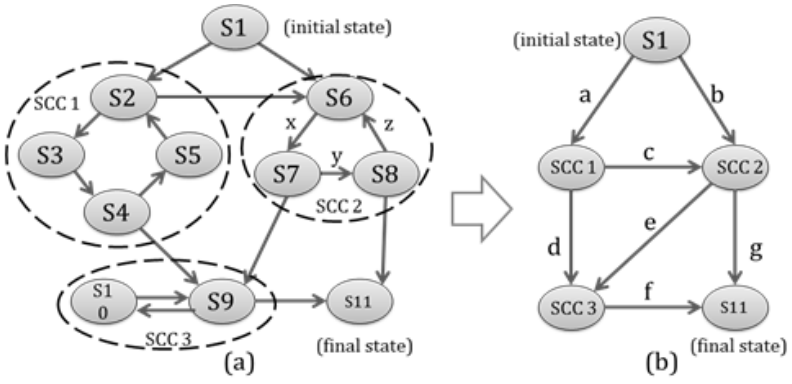


Figure 17 Example of SCC in an LTS graph

We point out some important properties for SCCs which will be used later: (1) There exists a path from a node to any other node in an SCC; (2) Any sequence of *t* events involved in an SCC is a valid t-way target sequence; (3) All the t-way target sequences in an SCC can be covered by a single transition sequence. Property (1) is derived from the definition of SCC. For property (2), given a sequence of *t* events, there exists a path that traverses *t* transitions corresponding to the t events in the given order and thus covers the given sequence of *t* events. Property (3) is the key point for the last step of this SCC-based algorithm. We denote a single transition sequence that covers all t-way target sequences in an SCC as a *t-touring* path, which can be generated using the following approach. First we build a 1-touring path P_1, i.e., a path that traverses all transitions *at least* once in an SCC. Note that it's different from an *Eulerian path* which requires every edge to be visited *exactly* once. Then starting from the last transition of P_1, we build another 1-touring path P_2 and append P_2 to P_1. By doing this, we make a 2-touring path. This process is repeated until we have a t-touring path. It can be verified that a t-touring path contains a sequence of *t* 1-touring paths, and therefore it covers all sequences of *t* events in the given SCC. For example, in Figure 17(a), a 3-touring path for SCC2 is *x•y•z•x•y•z•x•y•z* which covers all 3-way target sequences consisting of events *x,* y and z. A detailed algorithm for t-touring path generation is described in Figure 18.

B. Find Abstract Paths

The abstract paths are actually test sequences of the acyclic LTS *M'*, which can be generated using any test generation algorithm such as algorithm *GenTestSeqsBF* and *GenTestSeqsInc*.

Algorithm: **GenTouringPath**

Input: an SCC *M*, strength *t*, state *p,* state *q*

Output: a *t*-touring path *P* from state *p* to state *q*

1. find a shortest path between any two states in *M*
2. let *P* be an empty transition sequence
3. let state *s* = *p*
4. for (*i* = 1 to t) {
5. let *T* be all transitions in *M*
6. while (*T* is not empty){
7. let *r* be a transition in *T*
8. add into *P* a shortest path *H* from *s* to *src*(*r)*
9. remove from *T* any transitions traversed by *H*
10. let *s* be the last state of *P*
11. }
12. }
13. add into *P* a shortest path *H* from *s* to *r*
14. return *P*

Figure 18 An algorithm for t-touring path generation

C. Generate Test Sequences

As we discussed in step 2, we extend each abstract path by inserting a t-touring path after each transition ends with an SCC node, to extend it to a test sequence for M. Let S = s1 • s2 • …• sn be an abstract path of M'. If dest(si) is an SCC node, we insert a t-touring path P from dest(si) to src(si+1). In Figure 17(b), there are five abstract paths: a•d•f, a•c•e•f, a•c•g, b•e•f and b•g. Assume the test strength is 3. For path b•g, we insert a 3-touring path x•y•z•x•y•z•x•y•z•x•y after transition b, making a test sequence of the original LTS b•x•y•z•x•y•z•x•y•z•x•y•g. Similar insertions are made for other abstract paths. The whole SCC-based algorithm GenTestSeqsSCC is presented in Figure 19.

Algorithm: **GenTestSeqsSCC**

Input: an LTS M, a set Σ of target sequences, test strength t

Output: a t-way test sequence set Π for M

1. build an acyclic graph M' by finding and collapsing all SCCs in M
2. generate a set Ω consisting of abstract paths in M'
3. for (each abstract path $P = c_1 \bullet c_2 \bullet \ldots \bullet c_n$ in Ω){
4. for(i = 1 to n-1) {
5. if($dest(c_i)$ is an SCC node){
6. use algorithm *GenTouringPath* to generate a t-touring path T from $dest(c_i)$ to $src(c_{i+1})$,
7. insert T into P after transition c_i
8. }
9. }
10. if (the length of P is no less than t)
11. add P to Π
12. remove target sequences covered by P from Σ
13. }
14. if (Σ is not empty) {
15. use algorithm *GenTestSeqsFromTargets* to generate a test sequences set Π' to cover remaining targets
16. $\Pi = \Pi \cup \Pi'$
17. }
18. return Π

Figure 19 An SCC-base algorithm for test sequences generation

Complexity analysis: The complexity of this algorithm highly depends on the structure of graph. Assume that the original LTS contains n transitions. Finding SCCs takes O(n). Assume there are d transitions that end with an SCC nodes in m selected abstract paths, and there are in total k abstract paths. Assume each SCC contains n_s transitions. For each SCC, it takes O(tn_s) to build a t-touring path. The time complexity of line 2 is O($|\Sigma|mk$). There are d t-touring paths are inserted, therefore the total time is

O(dtn_s). The time complexity for the entire algorithm is O(n+|Σ|mk+dtn_s). The space complexity is O(|Σ|mk +dtn_s).

3.3.5 Comparison of Test Generation Algorithms

We provide an comparison of the four test sequence generation algorithms in Table 8.

Table 8 Comparison of Test Generation Algorithms

Algorithm	**Time cost**	**Space Cost**	**Length of individual test sequences**	**Number of generated test sequences**
GenTestSeqsFromTargets	low	low	short	many
GenTestSeqsBF	high	high	depends	few
GenTestSeqsInc	high	low	long	few
GenTestSeqsSCC	low	low	very long	very few

Algorithm *GenTestSeqsFromTargets* tends to generate many short test sequences, while algorithms *GenTestSeqsInc* and *GenTestSeqsSCC* tend to generate a small number of long test sequences. Algorithm *GenTestSeqsBF* has the highest time and space complexity but it can generate a small number of test sequences, when h is sufficiently large. Algorithm *GenTestSeqsSCC* has the lowest time and space complexity.

3.4 Experiments

We have built a tool that implements the proposed algorithms. To evaluate the performance of the proposed algorithms, we conducted experiments on both real and synthesized systems on a laptop with Core i5 2410M (2.30GHz) CPU and 4GB memory, running Windows 7 (64-bit) and Java 6 SE (32-bit) with the default heap size. The performance of the test generation algorithms is measured in terms of the total length of

generated test sequences, the number of generated test sequences, and the execution time taken to generate the test sequences.

3.4.1 Case Study: The Java Threads System

In this study we used the labeled transition system shown in Figure 11. Recall that this system describes the lifecycle of a Java thread, and it contains 7 states, 9 events and 16 transitions. The test strength is set to 3 in the experiments. There are 448 unique 3-way target sequences. Table 9 shows the results of applying the four test generation algorithms to this system. The columns in Table II are self-explanatory. For algorithm *GenTestSeqsInc*, we set *h*=5. For algorithm *GenTestSeqsBF*, we use *h*=16, 18, 20 in three different runs, since 16 is the critical length, i.e., the smallest *h* such that all target sequences will be covered by test sequences of length up to *h*.

Table 9 Result of the Java Threads System

JavaThreads (448 3-way targets)	**total length**	**num. of test seqs**	**avg. length**	**time (s)**
GenTestSeqsFromTargets	399	36	11.1	0.02
GenTestSeqsBF(h=16)	174	11	15.8	5.7
GenTestSeqsBF(h=18)	146	9	16.2	34.5
GenTestSeqsBF(h=20)	116	6	19.3	461.9
GenTestSeqsInc (h=5)	128	5	25.6	0.05
GenTestSeqsSCC	35	1	35.0	0.03

It is clear that, algorithm GenTestSeqsBF generates better test sequences as h increases, but execution time increases very fast. Algorithm GenTestSeqsSCC works very well for this system. In particular, it has the lowest number and total length of test sequences while running as fast as the fastest algorithm, i.e., algorithm GenTestSeqsFromTarget.

3.4.2 Synthesized Systems

In this study, we implemented a random LTS generator to generate synthesized systems. This generator takes three parameters, i.e., number of states, number of transitions, and number of events. The generation process consists of three major steps. First, it generates the given number of states and transitions, where each transition is placed between two random states. Second, it randomly assigns the given number of event labels to the transitions. Finally, it checks whether the generated graph is connected. A system is discarded if it is not connected.

We randomly generated 10 different systems, as shown in Table 10. Note that *SYS-n* denotes a system with *n* transitions. We also report the number of transitions which belong to any SCC in Table 10.

3.4.3 Results and discussions

Table 12 shows the test generation results of synthesized systems using four proposed algorithms. The test strength *t* is set to 3. Note that "-" indicates the process of test sequence generation took more than one hour to complete or ran out of memory. For algorithm *GenTestSeqsBF*, we set *h* to the critical length and increase it until timeout or out-of-memory. The *h* finally used in the first four systems is 30, 25, 15 and 13, respectively. For algorithm *GenTestSeqsInc*, we set *h*=5. For algorithm *GenTestSeqsSCC* we use *GenTestSeqsInc* to generate abstract paths, and the search depth is set to 10.

Table 10 Characteristics of Synthesized Systems

System	# of states	# of events	# of transitions	# of transitions in SCC
SYS-10	8	10	10	5
SYS-15	8	10	15	9
SYS-20	8	15	20	17
SYS-25	10	20	25	19
SYS-30	10	20	30	26
SYS-40	10	30	40	33
SYS-50	15	35	50	45
SYS-60	15	40	60	56
SYS-80	20	50	80	75
SYS-100	20	60	100	96

Table 11 Results of T-way Target Sequence Generation for Synthesized Systems

System	# of 2-way target seqs	# of 3-way target seqs	# of 4-way target seqs	# of 5-way target seqs
SYS-10	55	276	1380	6900
SYS-15	60	360	2160	12960
SYS-20	210	2940	41160	576240
SYS-25	303	4545	68175	1022625
SYS-30	360	6480	116640	2099520
SYS-40	750	18750	468750	11718750
SYS-50	1085	33635	1042685	32323235
SYS-60	1480	54760	2026120	74966440
SYS-80	2400	115200	5529600	265420800
SYS-100	3420	194940	11111580	633360060

Table 12 Results of test sequence generation for synthesized systems (3-way)

System	*GenTestSeqsFromTargets*				*GenTestSeqsBF*			
	total length	**# of test seqs**	**avg. length**	**time (s)**	**total length**	**# of test seqs**	**avg. length**	**time (s)**
SYS-10	184	16	11.5	0.01	55	3	18.3	4.1
SYS-15	273	26	10.5	0.01	80	4	20	9.6
SYS-20	5346	547	9.8	0.09	2083	164	12.7	84.5
SYS-25	5049	446	11.3	0.1	1534	104	14.8	693.9
SYS-30	8833	890	9.9	0.2	-	-	-	-
SYS-40	35432	4163	8.5	3.0	-	-	-	-
SYS-50	60047	5192	11.2	5.4	-	-	-	-
SYS-60	113391	11097	10.0	30.7	-	-	-	-
SYS-80	211975	18829	11.3	110.8	-	-	-	-
SYS-100	500022	47166	10.6	526.0	-	-	-	-

System	*GenTestSeqsInc*				*GenTestSeqsSCC*			
	total length	**# of test seqs**	**avg. length**	**time (s)**	**total length**	**# of test seqs**	**avg. length**	**time (s)**
SYS-10	123	11	11.1	0.01	55	3	18.3	0.01
SYS-15	297	22	13.5	0.03	84	4	21.0	0.08
SYS-20	477	16	29.8	1.3	74	1	74.0	0.2
SYS-25	1066	62	19.4	2.6	188	4	47.0	0.2
SYS-30	1381	55	25.1	32.5	132	2	66.0	0.4
SYS-40	7832	729	10.7	412.8	132	2	66.0	0.2
SYS-50	44058	1680	26.2	1080.7	439	4	109.8	2.4
SYS-60	-	-	-	-	403	3	134.3	6.2
SYS-80	-	-	-	-	354	2	177.0	10.2
SYS-100	-	-	-	-	557	3	185.7	14.6

Some observations can be made from these results. First, algorithm GenTestSeqsFromTargets is relatively fast, but it generates a large number of test sequences as well as the total length. Algorithm GenTestSeqsBF is very slow, but for small systems such as SYS-10 and SYS-15, it generates the best results in terms of the number of test sequences and total length. The GenTestSeqsInc algorithm achieves a good trade-off between total length and execution time. The GenTestSeqsSCC algorithm has very good performance, i.e., it generates very few test sequences in a very short time, and displays good scalability on large systems.

These algorithms have their own advantage and disadvantages, and can be used for different purposes and in different situations. Algorithm *GenTestSeqsFromTargets* is a useful strategy in general cases, and is preferred to be used in conjunction with other algorithms for covering remaining targets or speeding up the final stage. Algorithm *GenTestSeqsBF* can generate good test sequences for small systems. However the time and space complexity is very high and thus does not scale for large systems. Algorithm *GenTestSeqsInc* has small memory cost and reasonable performance. It is very flexible to work with various requirements. Algorithm *GenTestSeqsSCC* has the best scalability and performance among these algorithms and is suitable for larger systems. However, its performance highly depends on the structure of system graph. Furthermore, it generates long test sequences, which are not effective for fault localization, i.e., it's hard to identify which event sequence actually triggered the fault.

3.5 Related Work

In this section, we discuss related work in three areas, including combinatorial test data generation, combinatorial test sequence generation, and coverage criteria for test sequence generation.

3.5.1 Combinatorial Test Data Generation

Different strategies for combinatorial test data generation have been proposed in recent years. These strategies can be roughly classified into two groups, computational methods and algebraic methods [26] [27]. Computational methods usually involve an explicit enumeration of all possible combinations, and employ a greedy or heuristic search strategy to select tests. Examples of these methods include AETG [9], IPOG [5], and methods based on simulated annealing and hill climbing [27]. In contrast, algebraic methods build a t-way test set based on some pre-defined formulas, without enumerating any combinations. Examples of algebraic methods include orthogonal arrays [28] and doubling-construction [29]. A survey on combinatorial test generation methods can be found in [30]. Nonetheless, test data generation methods cannot be directly applied to test sequence generation, due to some fundamental differences between the two problems discussed earlier.

3.5.2 Test Sequence Generation

Several efforts have been reported that try to apply the idea of combinatorial testing to test systems that exhibit sequence-related behaviors. Wang *et al.* [31] [22] introduced a combinatorial approach to test dynamic web applications, which mainly investigates the fault detection effectiveness of a notion called pairwise interaction coverage on web applications. This is different from our work in that our focus is on efficient algorithms for test sequence generation. Kuhn *et al.* [32] presented several algorithms for t-way test sequence generation. Their algorithm requires constraints to be specified as prohibited sequences, and requires that each event occur exactly once in a test sequence. Thus, all the test sequences are of fixed length, which equals the total number of events. Our approach implicitly encodes constraints in a graph structure, and allows an event to occur an arbitrary number of times in a test sequence. Yuan *et al.*

introduced a covering array method for GUI test case generation in [33], and a more in-depth study on GUI testing [34]. Similar to the work in [31] [22], their work mainly investigates the fault detection effectiveness of different coverage criteria. The test generation methods are based on covering arrays, and all test sequences are extended from some fixed-length smoke tests. Our algorithms do not impose similar restrictions.

3.5.3 Test Sequence Coverage Criteria

We focus on coverage criteria that require "sequence of elements" to be covered. We do not consider coverage criteria such as all-nodes and all-branches [35]. Pairwise interaction coverage is used in some literatures such as Wang *et al.* [31] [22], which requires all possible pair of web page interactions to be covered by at least once. This is the special case of t-way sequence testing when t=2. Lucca and Penta [24] applied several coverage criteria for web application testing from [36]. One of their criteria is *all-transition-k-tuples*, which requires all possible sequence of *k* consecutive transitions to be covered. Similarly, a coverage criterion, called *length-n-event-sequence* coverage, was proposed for GUI testing [21]. These coverage criteria require a sequence of elements to be covered consecutively. This is different from our t-way sequence coverage criterion, which only requires a sequence of elements to be covered in the same order, not necessarily consecutively. The coverage criterion that is most closely related to ours is called *t*-coverage* [34]. This coverage is proposed for GUI interaction testing and requires all permutation of *t* events are executed consecutively at least once and inconsecutively at least once. In t-way sequence coverage proposed in this paper, a sequence of elements is considered to be covered as long as it is covered once, consecutively or inconsecutively.

Chapter 4

Case Study: Testing IEEE 11073 PHD

Personal healthcare is a rapidly growing market nowadays. Various personal healthcare devices such as weighing scales, blood pressure monitors, blood glucose monitors, and pulse oximeters have been developed in recent years. However, most of them cannot easily interoperate with each other. To address this problem, ISO/IEEE 11073 Personal Health Data (IEEE 11073 PHD) standards are developed to achieve interoperability between different personal healthcare devices. These standards are based on an earlier set of standards, i.e., ISO/IEEE 11073, which mainly focused on hospital medical devices. Compared to hospital medical devices, which are typically connected with an external power source, personal healthcare devices are normally portable, energy-limited, and have limited computing capacity. IEEE 11073 PHD standards adapt the earlier 11073 standards to take into account these unique characteristics of personal healthcare devices.

IEEE 11073 PHD has been adopted by many personal healthcare devices in the market. These devices typically have Bluetooth or ZigBee connectivity, and are able to transmit measured health data to healthcare professionals for remote health monitoring or health advising. To ensure that these products can interoperate with each other, it is important to ensure that these products conform to the standard communication behavior as specified by IEEE 11073 PHD. Many conformance testing procedures have been developed by organizations such as Continua Health Alliance [2].

As a case study of sequence testing, we propose a general conformance testing framework for the IEEE 11073-20601 protocol (Optimized Exchange Protocol) [3]. IEEE 11073-20601 is a core component in the standards family of IEEE 11073 PHD.

Specifically, IEEE 11073-20601 defines a communication model that allows personal healthcare devices to exchange data with computing resources like mobile phones, set-top boxes, and personal computers. The proposed framework consists of four major components, test sequence generator, test data generator, test executor, and test evaluator, each of which corresponds to a major step in the test process. The test generator adopts a technique called t-way testing, which has been shown to be a very effective software testing strategy [4] [5] [6]. The test executor is responsible for actually executing the tests generated by the test generator. The test evaluator is responsible for checking whether the actual outcome of each test execution is consistent with the expected outcome.

4.1 IEEE 11073 PHD STANDARDS

IEEE 11073 PHD defines an efficient data exchange protocol as well as the necessary data models for communication between two types of devices, i.e., agent and manager devices.

4.1.1 Agent and Manager

Agents are personal healthcare devices that are used to obtain measured health data from the user. They are normally portable, energy-efficient and have limited computing capacity. Examples of agent devices include blood pressure monitors, weighing scales and blood glucose monitors. Managers are designed to manage and process the data collected by agents. Examples of manager devices include personal computers, mobile phones and set top boxes. Manager devices are typically connected with an external power source. Data collected by agent devices can be used for further purpose like fitness advising, health monitoring and aging service provided by remote professionals. A typical scenario of using IEEE 11073 PHD personal healthcare devices and remote healthcare services is shown in Figure 20. In the left area of Figure 20, there

are various personal healthcare devices (agents) like blood pressure monitors, weighing scales and blood glucose monitors. Agents communicate with managers such as mobile phones, PCs, and set top boxes. The collected data can be sent to professionals for various remote services. IEEE 11073 PHD focuses the communication between agents and managers, as shown in the red box in Figure 20.

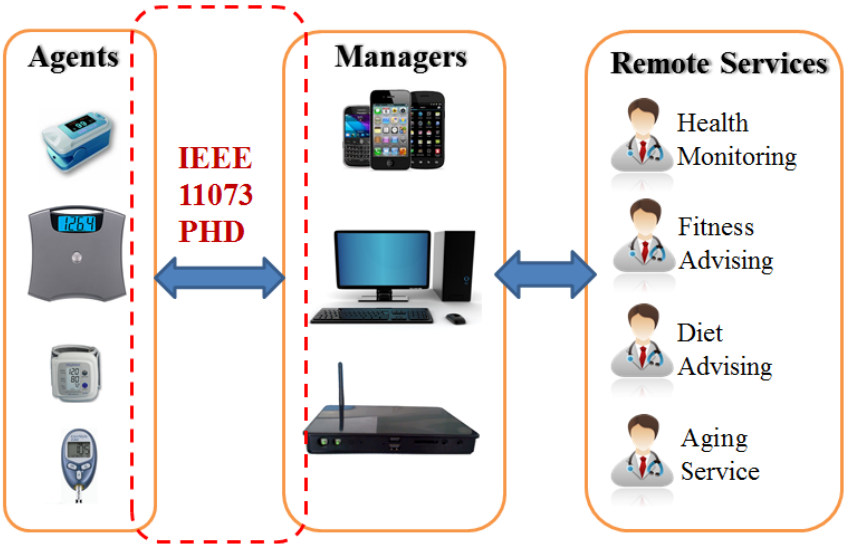


Figure 20 A Scenario of Using IEEE 11073 PHD Devices

IEEE 11073 PHD emphasizes the interoperability between various devices. That is, different devices should be able to communicate with each other out-of-box. In addition to data exchange, an agent device typically provides certain interface that allows a manager device to configure the device. For example, the operating frequency of a pulse oximeter can be adjusted by a manager device. IEEE 11073 PHD takes into account the different characteristics of agent and manager devices and treats them differently. In particular, communication between an agent and a manager is typically initiated and terminated by the agent when the measured data is available. This helps to reduce power consumption of the agent device as otherwise the agent would have to keep listening to incoming requests which could consume more energy. Also, since agent

devices typically have limited processing capability, they typically perform minimal data process and the data exchange between agent and manager devices is designed to be very concise.

4.1.2 Architecture

IEEE 11073 PHD consists of three major models, i.e., the domain information model (DIM), the service model, and the communication model, as shown in Figure 21. In the domain information model, a personal healthcare device is modeled as an object with multiple attributes. These attributes indicate configuration options, measured data and other particular functionalities. The service model defines data access procedures such as GET, SET, ACTION and Event-Report between agent and manager. For example, an agent is able to measure data and report it to the manager. On the other hand, the manager can configure certain agent attributes such as the frequency of operating. The communication model describes general point-to-point connection between an agent and a manager in terms of the agent state machines and the manager state machine.

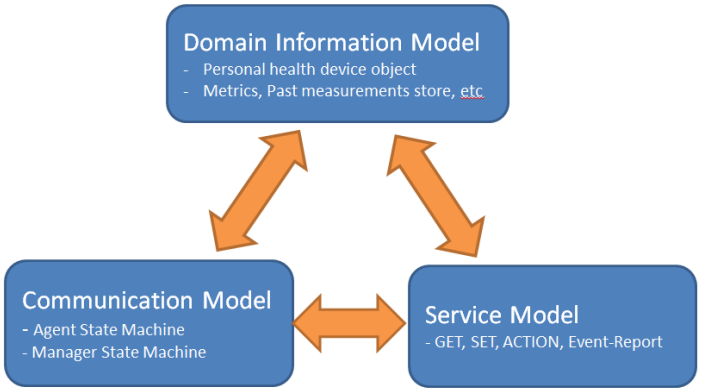


Figure 21 Three Major Models in IEEE 11073 PHD

4.1.3 IEEE 11073-20601

In this paper, we focus on IEEE 11073-20601, which is a core component of IEEE 11073 PHD. Multiple agents are allowed to establish connections to a single

manager. The point-to-point connection between an agent and a manager is independent with different transport layers such as Bluetooth, USB or ZigBee. The behavior of agent and manager are described in the agent state machine and the manager state machine, which is well defined in IEEE 11073-20601.

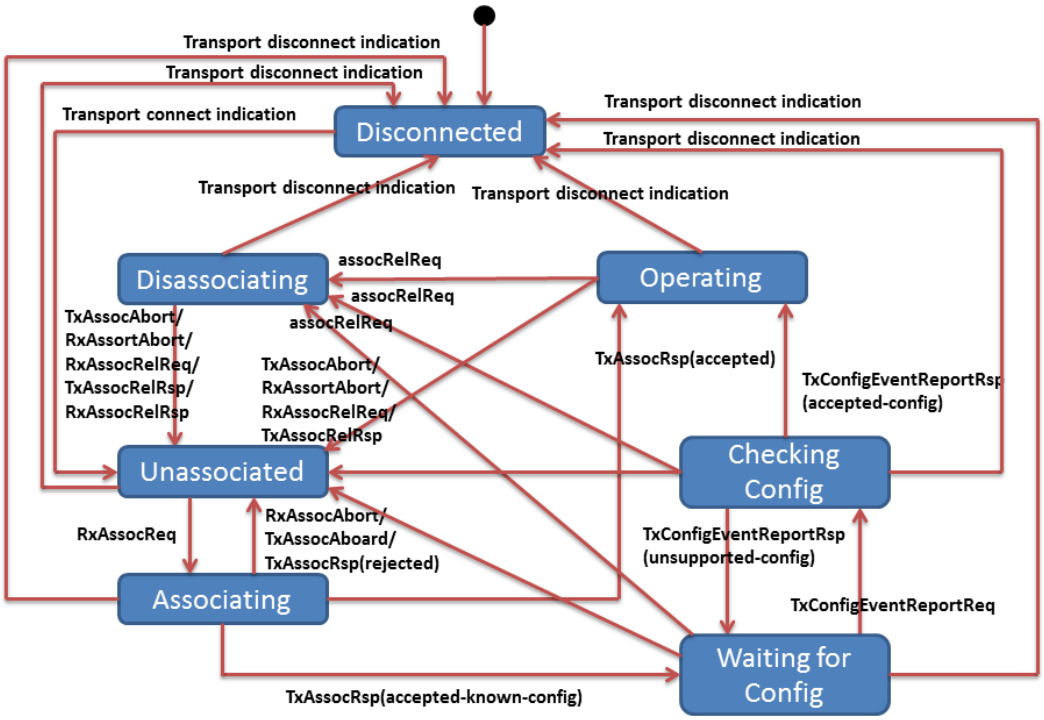


Figure 22 Manager State Machine (Flattened)

Figure 22 shows an overview of the manager state machine diagram. There are 7 states in this diagram, and we briefly introduce these states. When the personal healthcare device (agent) is turned on, it enters the *Disconnected* state and is ready to connect. When the transport connection between agent and manager is established, both agent and manager enter the *Unassociated* state. The agent then requires association with the manager by sending a request, and enters the *Associating* state. The manager will check the configuration of the agent, and then either accept this association request, or ask the agent for more information, or deny this request due to some failure or

unsupported protocol version. If the manager accepts association request, then both agent and manager enter the *Operating* states, and exchange data normally. Otherwise they enter the *Sending config* state and the agent needs to send the complete configuration profile to the manager, so that the manager can interoperate with the agent.

State transitions are triggered by specific events. For example, "*assocReq*" represents that the agent sends an association request to the manager. "*RxAssocRsp (accepted)*" represents that the agent receives the positive response of association from the manager. For more details about each event, one may refer to the protocol specification [3].

We use an example scenario to explain how the agent and manager exchange data. In Figure 23, the agent device is a weighting scale. It sends an association request to the manager. The association request contains the weighting scale's system ID, protocol version number and other device configurations. The manager may be configured to support certain devices. If the manager recognizes the system ID, it sends a response of association acceptance to the agent. Then both devices enter the *Operation* states. The agent sends a measurement data (weight) to the manager using a service type called "*Conformed Event Report*" defined in the service model. It requires the recipient to send an acknowledgement for that sent data. The manager successfully receives the weight value and sends back the acknowledgement. Finally the agent requests association release and the manager responds to this request. Both devices are now unassociated.

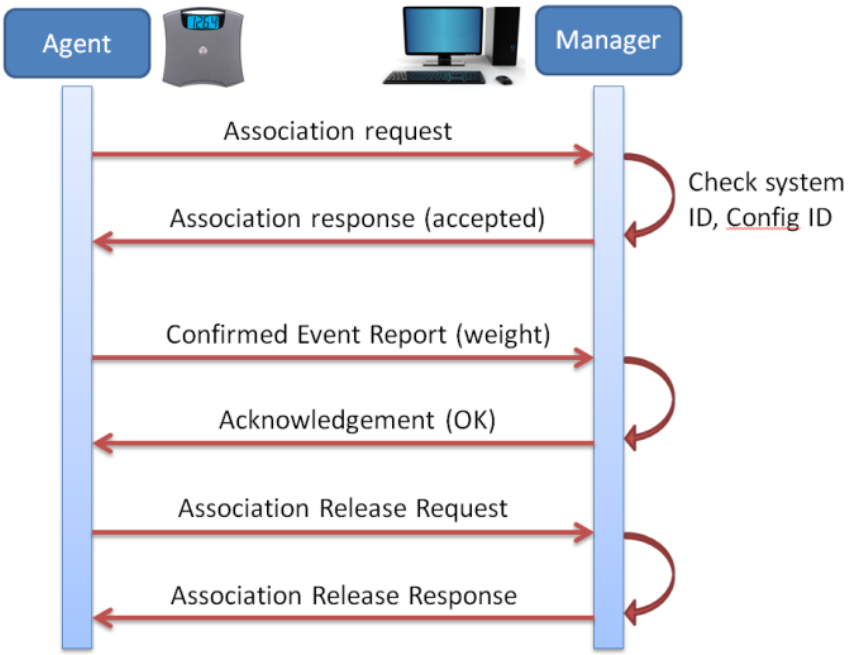


Figure 23 An Example Scenario of Data Exchange

4.2 The General Conformance Testing Framework

Figure 24 shows an overview of the proposed testing framework. The test sequence generator first generates test sequences from the state machine model as specified by IEEE 11073 PHD. Then the test data generator generates test data for each test sequence. The test data generator takes as input the domain information model which is supplied by the user. In the next step, the generated test sequences and data are executed by the test executor, and the test evaluator generates evaluation results from execution outcomes.

There are two state machines in the communication model. The agent state machine is maintained by the agent application, and the manager state machine is maintained by the manager application. We test them separately. A test driver is employed to interact with the agent or manager state machine that is being tested. When

we test an agent, the test driver acts like a manager. When we test a manager, the test driver acts like an agent.

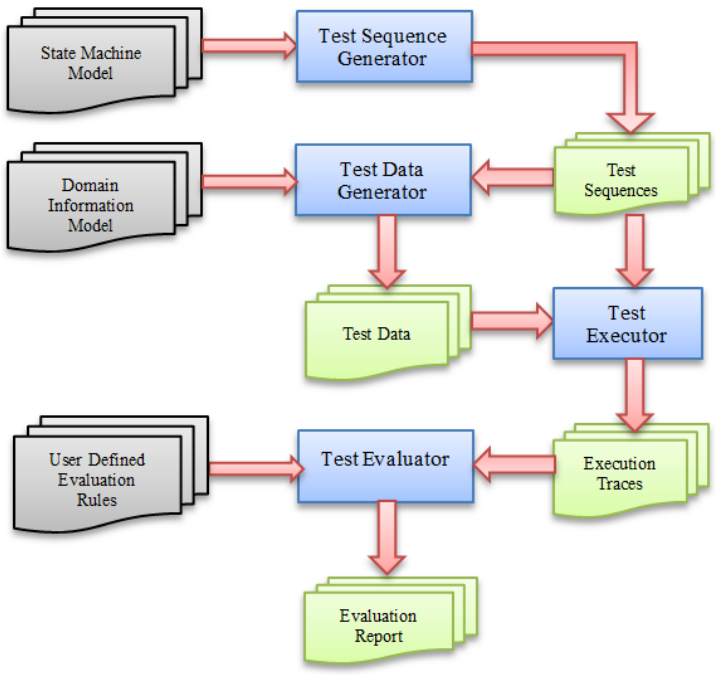


Figure 24 An Overview of the Proposed Framework

4.2.1 Test Sequence Generator

To apply sequence testing on IEEE 11073-20601, we first build two system models from the agent and manager state machine, respectively. The hierarchical structure of the state machines need to be flattened.

We generate test sequence using an existing test generation algorithm *GenTargetSeqs*. This algorithm builds test sequences from each target sequence that needs to be covered, and then select a small set of test sequences using a greedy algorithm. For the flattened agent LTS, we generated 249 test sequences for 2-way

testing. The same number of test sequences is generated for the manager state machine. The length of each test sequence ranges from 6 to 12.

4.2.2 Test Data Generator

A test sequence only specifies the types of the events that need to be exercised. To execute these events, test data must be generated for each test event. There are four types of events in the state machines, i.e., application requests, condition indications, receives event and sent events. Application requests like *association request* and *association release request* are triggered by higher level end-user application. Condition indications like *transport connect indication* and *transport disconnect indication* are triggered by lower level transports like TCP/IP connection. These events can be exercised by invoking certain API functions, and no data are needed. However, test data are need for executing the last two kinds of events.

For a send event, data is sent from the test driver. The test driver is responsible for constructing the concrete message, and sending it to the system under test. These messages are constructed according to information like message type, current state, and the domain information model, which is provided by the user though a XML file. For example, assume we are testing the manager state machine, and we need a send event *RxAssocReq*. This event means the agent sends a request of association. The test driver builds a message using user-provided information like device config-ID, and then sends it to the manager under test to execute this event.

For receive event, data is sent from the system under test itself. The message is constructed by the underlying protocol stack, thus only some system configuration like system ID are needed. For example, assume we are testing the agent, and it just sends an association request to the test driver (manager). The current state of agent is *Associating*. The next state could be either *Unassociated*, if the manager rejects the

association request; or be *Associated*, if the manager accepts this request. The decision is based on the configuration of agent. Thus in order to exercise the particular event, we have to generate correct configurations for the system under test.

4.2.3 Test Executor

The main challenge of performing sequence testing is how to execute the specific test sequences on the manager and agent, i.e., how to make events been actually exercised in order. As we discussed earlier, there are 4 types of events in state machines. These events are executed in different ways.

Application requests are normally triggered by the end-user applications, which interface with the state machine. How to execute these events highly depends on the application under test.

Condition indications come from low level software layer like transport plug-in. The standards require these indications to be triggered through well-defined APIs.

Send events are driven by the current state and certain conditions. These conditions are defined by the standard specifications. As discussed earlier, we generate test data for send events. If these data are correctly generated, these events will be exercised as expected.

Receive events are triggered by incoming messages sent from another application. If we are testing an agent device, we can use a manager application to communicate with it. Alternatively we can directly send the expected message to the agent. In this case, the test executor will read the data and then package them into messages, and then communicate with the system under test.

There are two methods to execute send/receive events, i.e., through transport layer or through API exposed to the user. The former method is more general, since we can use a standalone program to communicate with the system under test through

transport protocol such as USB, TCP/IP or Bluetooth. The latter one is more efficient since we invoke certain APIs to directly supply messages to the system under test, without network transmission. Since different IEEE 11073 applications may have different implementations, we provide a set of common interfaces for exercising protocol events. Then for different protocol implementations, we just use different adapters to trigger these events.

4.2.4 Test Evaluator

After the events are executed, we have to evaluate the result to check whether the device under test works as expected. First, we need to evaluate individual messages to check their format and verify certain data. Existing tools like ValidatePDU [7] developed by NIST can be used for this task. Second, we need to check if the messages are exchanged in the correct sequence. We check if each message received from the agent or manager state machine under test is expected. For example, the value *0xE200* in the message header indicates this message is an association request sent by the agent. We also have to check certain data field if user-defined evaluation rules are provided.

The evaluation results for each test sequence are aggregated and analyzed in order to generate a test report containing summaries and statistics of test execution, such as the number of correct event execution, the number of failed events, and the results for user-defined evaluation rules.

4.3 A Prototype Tool

A prototype tool is built by applying the proposed testing framework on Antidote, an open source implementation of IEEE 11073-20601. Antidote [1] is an open source implementation of IEEE 11073-20601. Antidote is a library that can be used to develop IEEE 11073 applications. The main design goal of Antidote is to provide a set of

convenient APIs that can handle communications for IEEE 11073 PHD Agent and Manager. The architecture of Antidote [8] is illustrated in Figure 25.

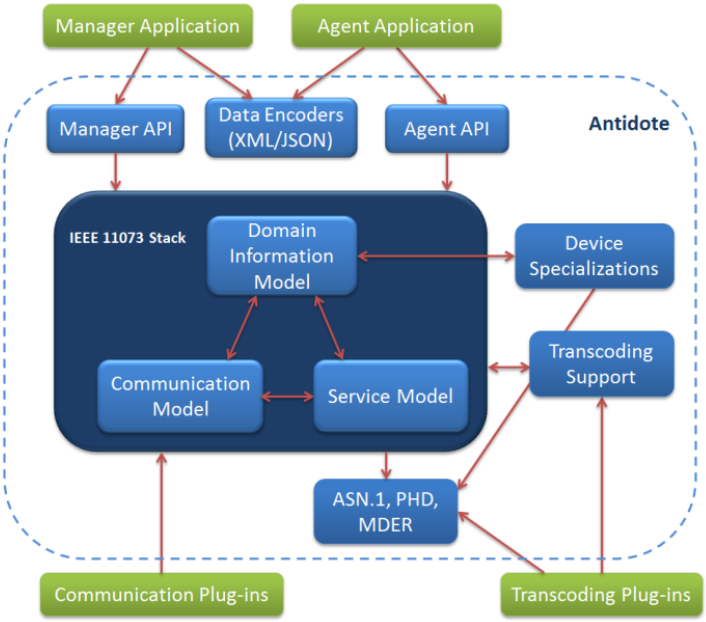


Figure 25 The Architecture of Antidote

In Figure 25, green components, i.e., manager application, communication plug-in and transcoding plug-in do not belong to Antidote implementation. They are implemented by the user for developing a real IEEE 11073 PHD application. Components in the dark blue area are IEEE 11073 PHD stack, including domain information model, service model and communication model, as we introduced earlier. They are well defined in the standards specification. Other components are Antidote specific components which facilitate the development for IEEE 11073 PHD applications. We briefly introduce some major components as follows. Manager and agent APIs include useful functions to the user for dealing with communication for IEEE 11073 PHD applications. Data encoders are used to encode data such as measurements and configuration in an independent format like XML and JSON, so that the developer does not need to be familiar with the

data format used in IEEE 11073 PHD. The communication plug-ins offer different choices for the transport. Antidote provides an interface for communication plug-ins, and allows the user to implement customized plug-ins. The transcoding plug-in allows devices that do not support IEEE 11073 PHD to communicate with Antidote.

As shown in Figure 24, the general framework has four major components, test sequence generator, test data generator, test executor, and test evaluator. Only the test executor component needs to be implemented in a way that is specific to Antidote, while the other components can be implemented in a way that is independent from Antidote. In the following, we focus on the implementation of the test executor for Antidote.

Test data are needed for executing test sequences. In order to execute each event in a test sequence, we have to invoke corresponding functions provided by Antidote, or communicate with the system under test using certain messages. Since in Antidote, the function for receiving messages is provided by the communication plug-in, we can construct messages and feed them directly to the system under test, without actually sending and receiving messages across a network.

We use an example to explain how to execute a test sequence. The test sequence for the manager state machine consists of 5 events. The state transition path is shown in Figure 26.

The first event "*Transport connect indication*" is triggered by a manager API function called "*manager_start*". By executing this function, the transport (TCP/IP in this example) is established. Then the agent requests association (*RxAssocReq*). For the manager, this event is an incoming message sent from the agent. The test driver then builds a correct message *msg*, and then sends it to the manager using function "*communication_process_apdu*" provided by the TCP/IP communication plug-in. Then the manager should accept this request automatically based on the agent configuration, if

we generated correct configuration data. A correct message containing “known configuration” will lead the manager to the *Operating* state, while an incorrect message containing “unknown configuration” will lead the manager to the *Waiting for configure* states. Then in the next event, the manager requests association release (*assocRelReq*) and this can be done by calling an API “*manager_request_association_release*” provided by Antidote. In the last step, the communication transport is disconnected by invoking another API “*manager_stop*”.

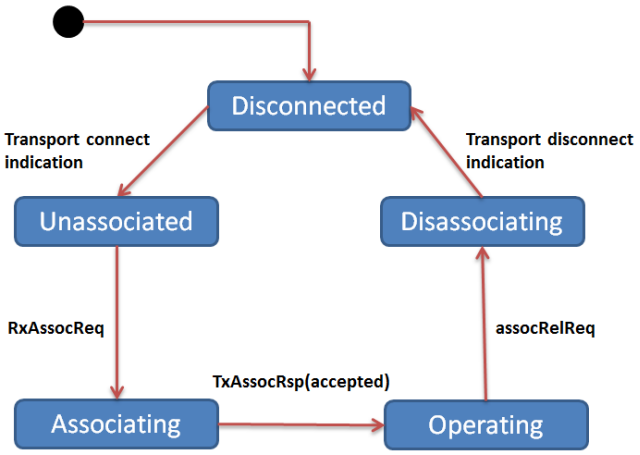


Figure 26 An Example of Transition Path (Manager)

4.4 Preliminary Results

In this section we report some preliminary results of testing Antidote using the prototype tool. We only tested the manager state machine. The flattened manager state machine shown in Figure 22 contains 7 states, 32 transitions and 15 unique events. Using the t-way sequence generation algorithm in [9], we generated 249 2-way test sequences with length ranging from 6 to 12. Each test sequence starts from the *disconnected* state, and ends with the same state. We executed these sequences and collected code coverage using a tool LCOV [10].

Since we focus on the communication model, we only present code coverage data for the source files in the *communication* folder of Antidote. This folder contains files that implement the communication model, in terms of state machines, transition rules, event handling, etc. The files in the sub-directories like *parser* and *plugin* are not counted, since they are not at the core of the communication model and thus are not the target of our experiment. Also we removed all functions related to agent operations since focus on manager state machine.

Current view: top level - src/communication		Hit	Total	Coverage
Test: coverage.info	Lines:	1016	1294	78.5 %
Date: 2013-06-05	Functions:	113	146	77.4 %

Filename	Line Coverage		Functions	
association.c	89.9 %	124 / 138	100.0 %	8 / 8
communication.c	85.8 %	297 / 346	91.5 %	43 / 47
configuring.c	88.1 %	119 / 135	77.8 %	7 / 9
context_manager.c	91.2 %	52 / 57	100.0 %	7 / 7
disassociating.c	54.3 %	25 / 46	50.0 %	3 / 6
extconfigurations.c	64.8 %	138 / 213	83.3 %	10 / 12
fsm.c	88.9 %	40 / 45	88.9 %	8 / 9
operating.c	63.2 %	103 / 163	40.9 %	9 / 22
service.c	72.2 %	83 / 115	60.0 %	12 / 20
stdconfigurations.c	97.2 %	35 / 36	100.0 %	6 / 6

Figure 27 Code Coverage Results

Figure 27 shows the code coverage results. For 6 files, i.e., *association.c*, *communication.c*, *configuring.c*, *context_manager.c*, *fsm.c*, and *stdconfigurations.c*, we achieved more than 80% coverage. Whereas the coverage results for other files are low, these results are consistent with or even better than our expectation because of the limited scope of our preliminary study. In particular, we only tested the implementation of the manager state machine, and we did not consider all events like error handling or operations defined in the service model. We emphasize that this is only a preliminary

experiment and it is our plan to test the entire implementation, including the agent state machine, the service mode, and the error handling mechanism.

4.5 Related Work

In recent years, many researches have been conducted on conformance and interoperability testing for medical/healthcare devices. These works can be divided into two categories, i.e., testing health information systems and testing medical or healthcare devices.

Snelick et. al. [11] compared conformance testing strategies for HL-7, a widely used standard for healthcare clinical data exchange. They analyzed two techniques for conducting conformance testing, i.e., using Upper Tester and Lower Tester, and using an actor based strategy. Namli [12] proposed a complete test execution framework for HL7-based systems. The framework is built up on an extensible test execution model. This model is represented by an interpretable test description language, which allows dynamic setup of test. Berube and Pambrun [13] presented a web application for testing interoperability in healthcare for sharing images between different institutions. Compared to their work, we focus on the communication model of IEEE 11073 PHD, and proposed a general framework for testing devices and applications that communicate using IEEE 11073 PHD standards.

Garguilo et. al. [7] developed conformance testing tools based on an XML schema derived directly from IEEE 11073 standard and corresponding electronic representations. The proposed conformance testing approach allows users to abstractly define devices using device profiles and implementation conformance statements. They are subsequently used to provide syntactic and semantic validation of medical device messages, according to IEEE 11073. This is complementary to our work. We focus on testing event sequences and their tool can be used to evaluate the execution results. Lim

et. al. [14] proposed a toolkit that can generate standard PHD messages using user-defined device information. This facilitates users who are not familiar with the standards details. This is similar to our test data generator, which generates individual messages from the domain information model.

Chapter 5

Conclusion and Future Work

This dissertation focuses on two important problems in combinatorial testing, including constrained test generation and combinatorial sequence testing.

For the first problem, we present an efficient algorithm, called IPOG-C, for constrained combinatorial test generation. The major contribution of our work is three optimizations employed by algorithm IPOG-C to improve the performance of constraint handling. These optimizations try to reduce the number of calls to a constraint solver. When such a call cannot be avoided, these optimizations try to reduce the number of constraints that have to be solved. We show that these optimizations can be applied to other test generation algorithms. Experiment results show that these optimizations can achieve performance improvements of up to two orders of magnitude. The IPOG-C algorithm is implemented in a combinatorial test generation tool, i.e., named ACTS, which is freely available to public. A comparative evaluation suggests that ACTS can perform significantly better than other tools for systems that have more complex constraints.

There are several directions to continue our work. First, we want to conduct more experiments to evaluate the effectiveness of our algorithm. In particular, the real-life systems in our experiments have a very small number of forbidden tuples. We want to investigate whether this is the case in general and if possible, apply our algorithm to real-life systems with a large number of forbidden tuples. Second, we want to develop efficient schemes to parallelize our algorithm. For example, we could divide the complete set of target combinations into several subsets, and then assign these subsets to different cores or processors. As another example, when we try to select the best value of a parameter, we could employ multiple cores or processors to determine the weight of each value. Finally, we plan to investigate how to integrate our algorithm into an existing

test infrastructure. Most work on combinatorial testing only addresses the test generation problem. Combinatorial testing can generate a large number of tests. It is thus particularly important to streamline the entire test process, i.e., integrate our test generation tool with other tools that automate test execution and test evaluation.

For the second problem, we presented our work on the problem of t-way test sequence generation. Our system model is defined in a general manner and can be used to model different types of systems, e.g., GUI applications, web applications and concurrent systems. We proposed an efficient algorithm for generating t-way target sequences that avoids redundant computations in checking the validity of all t-permutations of given events. We also presented several algorithms for generating test sequences to achieve t-way sequence coverage. We believe that these algorithms represent the first effort to systematically explore the possible strategies for solving the problem of t-way test sequence generation in a general context.

This work is the first stage of a larger effort that tries to expand the domain of combinatorial testing from test data generation to test sequence generation. In the next stage, we plan to conduct controlled experiments and case studies to investigate the fault detection effectiveness of t-way sequence testing for practical applications. In particular, we plan to apply and adapt the algorithms reported in this paper to test concurrent programs. Concurrency-related faults are notoriously difficult to detect because a concurrent program may exercise different synchronization behaviors due to the existence of race conditions. We believe that t-way sequence testing can be an effective technique to explore the different sequences of synchronization events that could be exercised by a concurrent program. Also we will complete our study of testing Antidote using the proposed framework. In particular, we will generate test sequences from the complete state machine, and also measure the effectiveness of the framework using real

and/or seeded faults in addition to code coverage. Furthermore, we will apply our framework to test some real devices to check their compliance with the IEEE 11073 PHD standards. The goal of our project is to develop a set of tools that can automate, as much as possible, the conformance testing process of medical devices designed to be IEEE 11073 PHD compliant.

References

[1] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of Experiments to Software Testing," in *27th NASA/IEEE Software Engineering Workshop*, 2002.

[2] D. R. Wallace and D. R. Kuhn, "Failure modes in medical device software: An analysis of 15 years of recall data," *International Journal of Reliability, Quality and Safety Engineering,* pp. 301-311, 2001.

[3] D. R. Kuhn, D. R. Wallace and A. J. Gallo Jr., "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering ,* vol. 30, no. 6, pp. 418-421 , 2004.

[4] M. Grindal, J. Offutt and J. Mellin, "Managing conficts when using combination strategies to test software," in *the 2007 Australian Software Engineering Conference (ASWEC 2007)*, 2007.

[5] Y. Lei, R. Kacker, D. Kuhn, V. Okun and J. Lawrence, "IPOG: A general strategy for t-way software testing," *Engineering of Computer-Based Systems,* 2007.

[6] ACTS. [Online]. Available: http://csrc.nist.gov/groups/SNS/acts/.

[7] M. Cohen, M. Dwyer and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *5th international symposium on software testing and analysis*, 2007.

[8] J. Czerwonka, "Pairwise testing in real world," in *10th Pacific northwest software quality conference*, 2006.

[9] D. M. Cohen, S. R. Dalal, M. L. Fredman and G. C. Patton, ""“The AETG system: An approach to testing based on combinatorial design," *IEEE Transactions On Software Engineering,* vol. 23, no. 7, p. 437–444, 1997.

[10] "Choco Solver," [Online]. Available: http://www.emn.fr/z-info/choco-solver/.

[11] M. Cohen, M. Dwyer and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach," *IEEE Transactions On Software Engineering,* vol. 34, p. 633–650, 2008.

[12] D. Kuhn and V. Okum, "Pseudo-exhaustive testing for software," in *IEEE/NASA Software Engineering Workshop*, 2006.

[13] A. Calvagna and A. Gargantini, "T-wise combinatorial interaction test suitees

construction based on coverage inheritance," in *Software Testing, Verification and Reliability*, 2009.

[14] B. Garvin, M. Cohen and M. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *1st International Symposium on Search Based Software Engineering*, 2009.

[15] B. Garvin, M. Cohen and M. Dwyer, "Evaluating Improvements to a Meta-Heuristic Search for Constrained Interaction Testing," *Empirical Software Engineering (EMSE),* vol. 16, no. 1, pp. 61-102, 2011.

[16] A. Hervieu, B. Baudry and A. Gottlieb, "PACOGEN : Automatic Generation of Test Configurations from Feature Models," in *Int. Symp. on Soft. Reliability Engineering*, 2011.

[17] G. Perrouin, S. Sen, J. Klein, B. Baudry and Y. L. Traon, "Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines," in *IEEE International Conference on Software Testing Validation and Verification*, 2010.

[18] M. F. Johansen, O. Haugen and F. Fleurey, "An Algorithm for Generating T-wise Covering Arrays from Large Feature Models," in *the 16th International Software Product Line Conference*, 2012.

[19] Y. S. Mahajan, Z. Fu and S. Malik, "Zchaff2004: An efficient sat solver," in *SAT 2004*, 2004.

[20] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill and D. Engler, "EXE: Automatically Generating Inputs of Death," *ACM Transactions on Information and System Security (TISSEC),* vol. 12, no. 2, 2008.

[21] A. M. Memon, M. L. Soffa and M. E. Pollack, "Coverage criteria for GUI testing," *the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering,* pp. 256-267, 2001.

[22] W. Wang, S. Sampath, Y. Lei and R. Kacker, "An Interaction-Based Test Sequence Generation Approach for Testing Web Applications," in *High Assurance Systems Engineering Symposium*, 2008.

[23] C. E. McDowell and D. P. Helmbold, "Debugging concurrent programs," *ACM Computing Surveys (CSUR),* p. 21(4):593–622, 1989.

[24] G. D. Lucca and M. D. Penta, "Considering Browser Interaction in Web Application

Testing," in *5th International Workshop on Web Site Evolution*, 2003.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, 2nd edition, The MIT Press, 2001.

[26] A. Hartman and L. Raskin, "Problems and algorithms for covering arrays," *Discrete Mathematics,* p. 149–156, 2004.

[27] M. B. Cohen, C. J. Colbourn, P. B. Gibbons and W. B. Mugridge, "Constructing test suites for interaction testing," in *25th IEEE International Conference on Software Engineering*, 2003.

[28] R. Mandl, "Orthogonal Latin squares: An application of experiment design to compiler testing," *Communications of the ACM,* 1985.

[29] M. A. Chateauneuf, C. J. Colbourn and D. L. Kreher, "Covering arrays of strength 3," in *Designs, Codes, and Cryptography*, 1999.

[30] M. Grindal, J. Offutt and S. F. Andler, "Combination Testing Strategies: A Survey," *Journal of Software Testing, Verification and Reliability,* pp. 97-133, 2005.

[31] W. Wang, Y. Lei, S. Sampath, R. Kacker, D. Kuhn and J. Lawrence, "A Combinatorial Approach to Building Navigation Graphs for Dynamic Web Applications," in *25th IEEE International Conference on Software Maintenance*, 2009.

[32] D. R. Kuhn, J. Higdon, J. Lawrence, R. Kacker and Y. Lei, "Combinatorial Methods for Event Sequence Testing," in *The 1st workshop on Combinatorial Testing in conjunction with the fifth International Conference on Software Testing*, 2012.

[33] X. Yuan, M. B. Cohen and A. M. Memon, "Covering Array Sampling of Input Event Sequences for Automated GUI Testing," in *22nd IEEE/ACM international conference on Automated Software Engineering*, 2007.

[34] X. Yuan, M. B. Cohen and A. M. Memon, "GUI Interaction Testing: Incorporating Event Context," *IEEE Transactions on Software Engineering,* 2011.

[35] A. Mathur, Foundations of Software Testing, Pearson Education, 2008.

[36] R. Binder, Testing Object-Oriented Systems, Addison Wesley, 2000.

[37] "Continua Health Alliance," [Online]. Available: http://www.continuaalliance.org/.

[38] *IEEE Std 11073-20601™, Health informatics – Personal health*

devicecommunication– Part 20601: Optimized exchange protocol.

[39] D. R. Kuhn, D. R. Wallace and A. J. Gallo Jr., "Software fault interactions and implications for software testing," *IEEE Transactions on Software,* 2004.

[40] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of," in *27th NASA/IEEE Software Engineering Workshop*, 2002.

[41] D. R. Wallace and D. R. Kuhn, "Failure modes in medical device software: An analysis of 15 years of recall data," *International Journal of Reliability, Quality and Safety Engineering,* 2001.

[42] J. Garguilo, S. Martinez and M. Cherkaoui, "Medical Device Communication: A Standards-based Conformance Testing Approach," in *the 9th International HL7 Interoperability Conference*, 2008.

[43] "Antidote Program Guide," 2012. [Online]. Available: http://oss.signove.com/index.php/File:AntidoteProgramGuide.pdf.

[44] L. Yu, Y. Lei, R. Kacker, D. R. Kuhn and J. Lawrence, "Efficient Algorithms for T-way Test Sequence Generation," in *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*, 2012.

[45] "LCOV: graphical GCOV front-end," [Online]. Available: http://ltp.sourceforge.net/coverage/lcov.php.

[46] R. Snelick, L. Gebase and M. Skall, "Conformance Testing and Interoperability: A Case Study in Healthcare Data Exchange," in *International Conference on Software Engineering Research and Practice*, 2008.

[47] T. Namli, G. Aluc and A. Dogac, "An Interoperability Test Framework for HL7-Based Systems, Information Technology in Biomedicine," in *IEEE Transactions on Information Technology in Biomedicine*, 2009.

[48] R. Berube and J. Pambrun, "Interoperability Testing Software for Sharing Medical Documents and Images," in *Fifth International Conference on Internet and Web Applications and Services*, 2010.

[49] J. Lim, C. Park, S. Park and K. Lee, "ISO/IEEE 11073 PHD message generation toolkit to standardize healthcare device," in *Engineering in Medicine and Biology Society*, 2011.

[50] D. Kuhn and M. Reilly, "An investigation of the applicability of design of experiments

to software testing," in *27th NASA/IEEE Software Engineering Workshop*, 2002.

[51] L. Yu, M. Nouroz Borazjany, Y. Lei, R. Kacker and D. R. Kuhn, "An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation," in *IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*, 2013.

[52] L. Yu, Y. Lei, R. Kacker and D. R. Kuhn, "ACTS: A Combinatorial Test Generation Tool," in *IEEE International Conference on Software Testing, Verification and Validation (ICST 2013 Tools Track)*, 2013.

[53] T. Thüm, C. Kästner, F. Benduhn and J. Meinicke, "FeatureIDE: An extensible framework for feature-oriented software development," *Science of Computer Programming,* 2012.

[54] M. N. Borazjany, L. Yu, Y. Lei, R. Kacker and R. Kuhn, "Combinatorial Testing of ACTS: A Case Study," 2012.

[55] D. Kuhn, J. Higdon, J. Lawrence, R. Kacker and Y. Lei, "Combinatorial Methods for Event Sequence Testing," in *FIrst Intl Workshop on Combinatorial Testing*, 2012.

[56] J. Lim, C. Park, S. Park and K. Lee, "ISO/IEEE 11073 PHD message generation toolkit to standardize healthcare device," in *IEEE Engineering in Medicine and Biology Society*, 2011.

[57] R. Berube and J. Pambrun, "Interoperability Testing Software for Sharing Medical Documents and Images," in *Fifth International Conference on Internet and Web Applications and Services*, 2010.

[58] M. Voelter, "Using Domain-Specific Languages for Product Line Engineering," Tutorial at SPLC 2009, 2009.

[59] T. Namli, G. Aluc and A. Dogac, "An Interoperability Test Framework for HL7-Based Systems, Information Technology in Biomedicine," *IEEE Transactions on Information Technology in Biomedicine,* vol. 13, no. 3, pp. 389- 399 , 2009.

[60] P. Brooks, B. Robinson and A. M. Memon, "An initial characterization of industrial graphical user interface systems," in *the 2nd IEEE International Conference on Software Testing, Verification and Validation*, 2009.

[61] M. Mendonca, M. Branco, Cowan and Donald, " S.P.L.O.T. - Software Product Lines Online Tools," in *In Companion to the 24th ACM SIGPLAN International Conference*

on Object-Oriented Programming, Systems, Languages, and Applications, 2009.

[62] M. Forbes, J. Lawrence, Y. Lei, R. Kacker and D. R. Kuhn, "Refining the In-Parameter-Order Strategy for Constructing Covering Arrays," 2008.

[63] J. Czerwonka, "Pairwise testing in real world," in *10th Pacific northwest software quality conference*, 2006.

[64] K. Pohl, G. Böckle and F. J. v. d. Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag New York, Inc., 2005.

[65] A. W. Williams, "Determination of test configurations for pair-wise interaction coverage," in *13th International Conference on the Testing of Communicating Systems*, 2000.

[66] Y. Tung and W. S. Aldiwan, "Automating test case generation for the new generation mission software system," in *IEEE Aerospace Conference*, 2000.

[67] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak and A. S. Peterson, "Feature-oriented domain analysis," Carnegie-Mellon University Software Engineering, 1990.

[68] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers,* 1986.

[69] S. Warshall, "A theorem on boolean matrices. Journal of the ACM," *Journal of the ACM,* 1962.

[70] "Antidote Program Guide," [Online]. Available: http://oss.signove.com/index.php/File:AntidoteProgramGuide.pdf.

[71] [Online]. Available: http://oss.signove.com/index.php/Antidote:_IEEE_11073-20601_stack.

Biographical Information

Linbin Yu received his B.S. degree in Electronic Science and Technology and M.S degree in Computer Science from the University of Science and Technology of China in 2004 and 2009 respectively. He received his PhD degree in Computer Science and Engineering from the University of Texas at Arlington in August 2013.