ADVANCED ALGORITHMS FOR COMBINATORIAL AND SEQUENTIAL TEST GENERATION WITH CONSTRAINTS

by

FENG DUAN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy at
The University of Texas at Arlington
May, 2021

Arlington, Texas

Supervising Committee:

Yu Lei, Supervising Professor
Christoph Csallner
Hao Che
Jiang Ming

ABSTRACT

Advanced Algorithms for Combinatorial and Sequential Test Generation with Constraints

Feng Duan, Ph.D.

The University of Texas at Arlington, 2021

Supervising Professor: Jeff (Yu) Lei

Combinatorial and sequential testing are software testing strategies that have attracted significant interests from both academic and industrial communities. This dissertation addresses the problem of how to efficiently generate tests for both combinatorial and sequential testing.

The dissertation makes two major contributions. For combinatorial testing, we present several optimizations on an existing *t*-way test generation algorithm called IPOG. These optimizations are designed to reduce the number of tests generated by IPOG. For sequential testing, we develop a notion for expressing commonly used sequencing constraints and present a *t*-way test sequence generation algorithm that support constraints expressed using notation. We demonstrate the effectiveness of our notation and test generation algorithm using a real-life protocol that exhibits sequencing behavior.

This dissertation is presented in an article-based format, including three published research papers and one manuscript that is currently under review. The first two papers are about

combinatorial test generation. The other two papers are about sequential test generation.

The first paper reports our work on reducing the number of tests generated during the vertical growth phase of the IPOG algorithm, where the vertical growth problem is modeled as the classical "Minimum Vertex Coloring" problem. The second paper reports our work on extending the approach in the first paper to support constraints, where constraints are represented as edges and hyperedges in a graph structure.

The third paper reports our work on addressing the problem of constraint handling in sequential testing, where we design a notation for sequencing constraint specification and develop a new algorithm to handle constraints expressed using our notation. The fourth paper reports our latest work on sequential testing, where we translate constraints expressed using our notation to Deterministic Finite Automaton (DFA) and use DFA to check sequence validity and extensibility.

ACKNOWLEDGEMENTS

I would like to express the deepest appreciation to my supervising professor Dr. Jeff (Yu) Lei, whose expertise was invaluable in formulating the research questions and methodology. Without his supervision and constant help this dissertation would not have been possible.

I would like to thank my committee members, Dr. Christoph Csallner, Dr. Hao Che and Dr. Jiang Ming for generously sharing their time and ideas.

In addition, I would like to appreciate my parents for their constant love and patience. I am grateful to my younger brother for his support. Finally, I wish to give my heartfelt thanks to my wife whose unconditional love and continual support enabled me to complete this dissertation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1.

INTRODUCTION

1.1 Research overview

In this dissertation, we present our research on advanced algorithms for both combinatorial and sequential test generation supporting constraints.

Combinatorial and sequential testing are software testing strategies that have attracted significant interests from both academia and industry. The basic concept of combinatorial and sequential testing is that most software failures are caused by interactions of only a few parameters or events. Combinatorial testing has been widely accepted as a way of detecting *t*-way interaction failures as two or more parameter values interact to cause the program to reach an incorrect result. The interaction level *t* is often referred to as *strength*. This concept of *t*-way testing has then been expanded from parameter values to event orders, called *t*-way sequence testing.

Many systems such as interactive systems, event-driven systems and communication protocols, exhibit sequencing behavior, where a sequence of events is exercised during each execution and the order in which the events occur could significantly affect the system behavior. To test these systems, we need to generate test sequences, in addition to test data as parameter values. T-way sequence testing applies the notion of *t*-way coverage to test sequence generation.

If a test set contains every possible combination of values of any *t* parameters, we say that it achieves *t*-way combinatorial coverage (abbr. *t*-way coverage). Similarly, if a test sequence set covers every possible sequence of *t* events, we say that it achieves *t*-way sequential coverage (a.k.a. *t*-way sequence coverage). Informally, given any *t* events, if they could be exercised in a specific order, there must exist at least one test sequence in which these events are exercised in the same order (not necessarily consecutively). Doing so allows us to test all possible interactions among any *t* events. Thus, *t*-way sequence testing can expose faults that are caused by interactions among no more than *t* events.

One challenge of combinatorial (and sequential) testing is to generate a test set that is as small as possible, i.e., optimal test set, while achieving *t*-way coverage. To generate optimal test set is a trade-off between the time cost of test generation and the effort of test execution and evaluation. In other words, it would spend more time to generate an optimal test set than arbitrary one, while optimal test set can save the effort of tester to execute tests and evaluate test results.

A second challenge is that combinatorial and sequential test generation algorithms must handle constraints. Constraints are relationships among the parameters (or events) of System Under Test (SUT) that forbid certain values of some parameters from appearing in the same test case (or forbid some event sequences from appearing in the same test sequence). We focus on positive testing in this dissertation. That is, we generate tests that satisfy all the constraints. Note that constraints are also useful for negative testing where tests could be generated to violate constraints in a systematic manner.

For combinatorial testing, we present our work on optimizing an existing algorithm for combinatorial test generation called In-Parameter-Order-General (IPOG), which is a generalization of a 2-way (a.k.a. pairwise) test generation algorithm called In-Parameter-Order (IPO). For a system with t or more parameters, the IPOG algorithm builds a t-way test set for the first t parameters, extends the test set to build a t-way test set for the first t+1 parameters, and then continues to extend the test set until it builds a t-way test set for all the parameters. The extension of an existing t-way test set for a new parameter is performed in two phases: (1) Horizontal growth, which extends each existing test by adding one value for the new parameter; (2) Vertical growth, which adds, if needed, new tests to cover missing tuples, i.e., tuples that have not been covered yet.

The challenge of vertical growth is to minimize the number of new tests. The vertical growth part of the IPO algorithm is optimal for 2-way combinatorial test generation without constraints. When IPO is generalized for t-way testing, the vertical growth part is modified in a straightforward manner making the vertical growth part of the IPOG algorithm not optimal for general t-way test generation with or without constraints.

To meet this challenge, we first reduce the vertical growth problem to a classical graph problem called “Minimum Vertex Coloring”. We represent missing tuples as uncolored vertices, and the existing tests as colored vertices with distinct colors. Thus, the minimum number of colors with which the vertices may be colored, called the chromatic number, is the optimal test set size of vertical growth. Based on this reduction, we develop an algorithm that improves the vertical

growth part of the IPOG algorithm. We consider both *compatibility* and *validity conflicts*: (1) compatibility conflicts are derived from assigning two different values on the same parameter, i.e., edges between two missing tuples, or between one existing test and one missing tuple; (2) validity conflicts are derived from constraints. A validity conflict is difficult to handle as it may involve more than two vertices and thus cannot be represented as an edge in a graph.

We use the notion of hyperedge to represent validity conflicts and reduce the vertical growth problem to a hypergraph coloring problem. A hyperedge is an edge that may involve more than two vertices. A hypergraph is a generalization of a graph in which an edge can join any number of vertices. We develop an approach to create hyperedges representing validity conflicts from constraints: (1) We transform constraints into a set of Minimum Forbidden Tuples (MFTs) (A forbidden tuple is a tuple consisting of values of some parameters that violates constraints. A MFT is a forbidden tuple of minimum size that contains no other forbidden tuples); (2) For each MFT, we enumerate combinations of vertices that cover all parameter values of the MFT and represent them as hyperedge candidates (Hyperedge candidates are formed by at least two missing tuples, or one existing test and at least one missing tuple); (3) A hyperedge candidate becomes a hyperedge if it is compatible (i.e., it does not contain compatibility conflict) and minimum (i.e., it does not properly contain other hyperedges).

Edges that represent compatibility conflicts and hyperedges that represent validity conflicts are then used to calculate the Degree of Conflicts (DOC) for each uncolored vertex which may be involved in a certain number of edges and hyperedges. A greedy hypergraph coloring method is

deployed to color the uncolored vertices in the non-increasing order of DOC, i.e., cover tuples which involve more conflicts earlier. The experimental results show that the optimized IPOG algorithm can generate optimal test sets for many real-life systems with constraints.

For sequential testing, we present our work on the design of an event-oriented notation for specifying sequencing constraints and the development of a *t*-way test sequence generation algorithm supporting constraints specified by our notation.

As mentioned above, one important problem in sequential testing is dealing with sequencing constraints, i.e., restrictions on the order of events that must be satisfied for a test sequence to be valid. The technical challenge is two-fold. First, a notation is needed to specify sequencing constraints. This notation must be easy to use and have the power to express commonly encountered constraints. Second, a test generation algorithm must be developed to handle sequencing constraints. Compared to constraints on parameter values, sequencing constraints can be more difficult to handle. This is because the search space of possible solutions for sequencing constraints can be much larger due to the extra dimension, i.e., order of events. Note that besides sequencing constraints, we also define repetition and length constraints to control the length of a test sequence.

Our notation adopts an event-oriented framework which defines a small set of operators that capture fundamental order restrictions that could happen between two events. These operators can be nested, if necessary, to specify the sequencing behavior among multiple events. Our notation is at a higher level of abstraction than an operational model such as Finite State Machine (FSM). We

believe that while we are dealing with sequences of events, an event-oriented notation is more intuitive than a state-machine-based notation.

Our algorithm employs a greedy strategy in which each test sequence is generated to cover the maximal number of *t*-way target sequences. A *t*-way target sequence is a sequence of *t* events that can be covered by a test sequence in the given order (not necessarily consecutively), i.e., the order in which they appear in the sequence. Each test sequence is generated in two phases, including the starting phase and the extension phase. In the starting phase, we generate a starting sequence that is guaranteed to cover at least one target sequence. In the extension phase, we keep extending the test sequence until no extension is possible. At each extension, we append to the test sequence an event that covers the most *t*-way target sequences that are yet to be covered.

In order to improve the performance of sequencing constraint handling, we optimize our generation algorithm by two steps: (1) We formalize the translation from our notation to Deterministic Finite Automaton (DFA) via automata operations; (2) We perform sequence validity and extensibility check using DFA which can significantly reduce time complexity. We apply our approach to a real-life communication protocol "IEEE 11073-20601". With its latest set of 9 sequencing constraints expressed using our notation, we generate sets of test sequences that achieve 2-way and 3-way sequence coverage while satisfying all the constraints. The experiment results show that our approach is practical for *t*-way sequential testing.

1.2 Summary of publications

This dissertation is presented in an article-based format, including three published research papers and one manuscript that is currently under review.

Chapter 2 presents the paper titled, “Improving IPOG's Vertical Growth Based on a Graph Coloring Scheme”, which was published in IEEE 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), in 2015. Note that this paper is co-authored which includes the following authors besides me: Yu Lei, Linbin Yu, Raghu N. Kacker, and D. Richard Kuhn. I am the primary author of this paper and take the lead of this project; Yu Lei is the supervisor of this project; and all the rest of co-authors contribute to the revise of this paper.

This paper reports our work on reducing the number of tests generated during the vertical growth phase of IPOG algorithm, which is optimal for t-way test generation without constraints when $t = 2$ but no longer optimal when t is greater than 2. The vertical growth problem is modeled as a classical NP-hard problem called “Minimum Vertex Coloring”; then a greedy coloring approach is adopted to determine the order in which missing tuples are covered during vertical growth. The experimental results show that, compared with the original IPOG algorithm which uses an arbitrary order to cover missing tuples during vertical growth, the revised IPOG algorithm reduces the number of tests for many real-life systems.

Chapter 3 presents the paper titled, “Optimizing IPOG's Vertical Growth with Constraints Based on Hypergraph Coloring”, which was published in IEEE 10th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), in 2017. Note that this paper is co-authored which includes the following authors besides me: Yu Lei, Linbin Yu, Raghu N. Kacker, and D. Richard Kuhn. I am the primary author of this paper and take the lead of this project; Yu Lei is the supervisor of this project; Linbin Yu contributes to the idea of MFT which was presented in his prior work; all the rest of co-authors contribute to the revise of this paper.

This paper presents our work on the expansion from graph coloring approach in Chapter 2 to hypergraph coloring which supports constraints. In the hypergraph model, vertices are either missing tuples waiting to be colored or existing tests already colored in distinct colors at the initial state; edges/hyperedges are conflicts among vertices that cannot be put in a same test. After coloring, a group of vertices in same color can be transformed to exactly a valid test. In the new IPOG algorithm incorporating this optimization, Degree of Conflicts (DOC) should be computed for each tuple; then the tuples would be covered in the non-increasing order of DOC. The experimental results show that it reduces the number of tests for many real-life systems with constraints.

Chapter 4 presents the paper titled “An Approach to T-way Test Sequence Generation With Constraints”, which was published in IEEE 12th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), in 2019. Note that this paper is co-authored which includes the following authors besides me: Yu Lei, Raghu N. Kacker, and D. Richard Kuhn.

I am the primary author of this paper and take the lead of this project; Yu Lei is the supervisor of this project; and all the rest of co-authors contribute to the revise of this paper.

In this paper we address the problem of constraint handling in *t*-way test sequence generation. We design a notation to specify sequencing constraints and develop an algorithm of *t*-way test sequence generation to handle the constraints specified using this notation. We report a case study in which our notation and generation algorithm are applied to a real-life communication protocol IEEE 11073-20601 (Optimized Exchange Protocol). Our experience indicates that our notation is intuitive to use and allows us to express important sequencing constraints for the protocol. However, the experiment results show that our generation algorithm takes a significant amount of time, which requires further effort to make *t*-way sequence testing practically useful.

Chapter 5 presents the manuscript including our further research on the topic of sequential testing, titled "T-way Test Sequence Generation using an Event-Oriented Notation", which is submitted to the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), in 2021. Note that this manuscript is a co-authored work which also includes the following authors besides me: Xiaolei Ren, Yu Lei, Raghu N. Kacker, and D. Richard Kuhn. I am the primary author and project leader; Yu Lei is the supervisor of this project; and all the rest of co-authors contribute to the revise of this manuscript.

In this manuscript we address the time cost problem of *t*-way sequence generation to make it practical. We amend our notation for sequencing constraint specification, in order to accurately define the nesting structure of complex constraint and clearly describe how to translate a

constraint from our notation to Deterministic Finite Automaton (DFA) based on automata operations. We optimize the sequencing constraint handler by using DFA to perform validity and extensibility check on sequences, which greatly reduce the time cost of *t*-way test sequence generation with constraints. We analyze the generation results to find a local optimal set of test sequences that consists of the least number of sequences and has the minimum sum of sequence lengths.

Finally, in Chapter 6, we provide the conclusion of our research and discuss possible directions for our future work.

IMPROVING IPOG'S VERTICAL GROWTH BASED ON A GRAPH COLORING SCHEME

Feng Duan, Yu Lei, Linbin Yu, Raghu N. Kacker, and D. Richard Kuhn. “Improving IPOG's Vertical Growth Based on a Graph Coloring Scheme.” In 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1-8. IEEE, 2015.

CHAPTER 2.

Improving IPOG's Vertical Growth Based on a Graph Coloring Scheme

Abstract—We show that the vertical growth phase of IPOG is optimal for *t*-way test generation when $t = 2$, but it is no longer optimal when t is greater than 2. We present an improvement that reduces the number of tests generated during vertical growth. The vertical growth problem is modeled as a classical NP-hard problem called “Minimum Vertex Coloring”. We adopted a greedy coloring algorithm to determine the order in which missing tuples are covered during vertical growth. We implemented a revised IPOG algorithm incorporating this improvement. The experimental results show that compared with the original IPOG algorithm, which uses an arbitrary order to cover missing tuples during vertical growth, the revised IPOG algorithm reduces the number of tests for many real-life systems.

Keywords—*Combinatorial testing; Multi-way test generation; ACTS; Minimum vertex coloring; Tuple ordering*

2.1 Introduction

In our earlier work, we developed a *t*-way test generation algorithm called In-Parameter-Order-General (IPOG) [3-5]. The IPOG algorithm is a generalization of a pairwise or 2-way test generation algorithm called In-Parameter-Order (IPO) [1, 2]. For a system with t or more parameters, the IPOG algorithm builds a *t*-way test set for the first t parameters, extends the

test set to build a *t*-way test set for the first *t+1* parameters, and then continues to extend the test set until it builds a *t*-way test set for all the parameters.

The extension of an existing *t*-way test set for an additional parameter is done in two phases:

- Horizontal growth, which extends each existing test by adding one value for the new parameter;
- Vertical growth, which adds new tests, if needed, to cover the remaining tuples that have not been covered yet.

A careful examination of the IPOG algorithm reveals that the vertical growth phase is optimal for *t*-way test generation when $t = 2$, but it is no longer optimal when $t > 2$. This is partly because the IPO algorithm was originally designed for 2-way testing. When the IPO algorithm was generalized, no effort was made to optimize the vertical growth phase, and a straightforward extension was adopted.

In this paper, we present an improvement that reduces the number of tests generated during vertical growth. The vertical growth problem is modeled as a classical NP-hard problem called "Minimum Vertex Coloring" [11, 12]. We adopt a greedy coloring algorithm to determine the order in which missing tuples are covered during vertical growth. This is in contrast with the current algorithm where tuples are covered in an arbitrary order, i.e., as they are encountered, during vertical growth. In this paper we focus on *t*-way test generation without constraints, leaving constraint handling as part of our future work.

We implemented a revised IPOG algorithm that incorporates the above improvement in ACTS [6, 7]. We conducted experiments on a set of real-life systems that have been used to evaluate the effectiveness of *t*-way test generation algorithms. The experimental results show that the revised algorithm performed better than the original IPOG algorithm implemented in ACTS, and better than PICT [8, 9], for a set of real-life systems.

The remainder of the paper is organized as follows. Section 2.2 briefly reviews the IPOG algorithm for *t*-way testing. Section 2.3 describes two motivating examples to show why the vertical growth phase of IPOG for *t*-way test generation when $t > 2$ is not optimized, while it is optimal for 2-way test generation. Section 2.4 presents the improvement based on a graph coloring scheme. Section 2.5 reports the design and the results of the experiments. Section 2.6 discusses related work improving IPOG. Section 2.7 provides concluding remarks and our plan for future work.

2.2 The IPOG Algorithm

In this section, we present the major steps of IPOG algorithm, as shown in Figure 2-1. Refer to [3] for more details.

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 (also known as tuples) 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.

```
Algorithm IPOG(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 Pn
3. add into test set ts a test for each combination of values of the first t parameters
4. for (int i = t + 1; i ≤ n; i ++){
5.      let π be the set of all t-way combinations of values involving parameter Pi, and any group of (t-1) parameters
        among the first i-1 parameters
6.      // horizontal extension 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' 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 extension 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 test τ' that covers both τ and σ , i.e., τ is compatible
            with σ) {
14.             replace test τ with τ' in ts
15.             remove from π the combinations of values covered by τ'
16.         } else {
17.             add a new test τ only contains σ into ts
18.             remove from π the combinations of values covered by τ
19.         } // end if at line 13
20.     } // end for at line 12
21. } // end for at line 4
22. return ts
}
```

Figure 2-1. The original version of the IPOG algorithm

When we use a test to cover a combination, only “don’t care” values can be changed. A “don’t care” value is a value that can be replaced by any value without affecting the coverage of a test set. If no existing test can be changed to cover a combination σ, a new test needs to be added in which the parameters involved in σ are assigned the same value as in σ and the other parameters are assigned “don’t care” values.

The new parameter, i.e., the parameter to be covered by the current extension of the existing test set, is assigned the “don’t care” value in a test when choosing any possible value of this new parameter does not cover any new combination in the test. This only happens during horizontal growth. The old parameters, i.e., the parameters that have already been covered by the existing test set, are assigned the “don’t care” value during vertical growth when adding a new test to cover a combination. Some of these “don’t care” values will be changed to a specific value at a later point of vertical growth for the same parameter whereas others may survive to the extension for the next parameter to be covered.

Thus, there may exist three types of test with “don’t care” values in the existing test set after horizontal growth:

1. *Tests only have “don’t care” value for the new parameter:* No more (uncovered) combinations can be covered, since horizontal growth has tried all possible extensions for this test using all possible values of the new parameter.

2. *Tests only have "don't care" values for the old parameters:* Though the value of the new parameter has been determined, there may still exist some possible extensions, since horizontal growth does not consider all possible values of the old parameters.
3. *Tests have "don't care" values for the new parameter and the old parameters:* Choosing any value of the new parameter would not cover any more uncovered combinations. But there may exist some possible extensions for the "don't care" values of the old parameters and the new parameter combined together during vertical growth.

Types 2 and 3 indicate opportunities where uncovered combinations can be covered by existing tests, i.e., without adding new tests. The challenge is that, for a general system, if there is a systematic strategy for changing "don't care" values to reduce the number of tests as much as possible.

2.3 Motivating Examples

Assume that the horizontal growth phase of IPOG has been finished for a given system, and the vertical growth phase is about to begin. Since the new parameter is involved in every missing tuple, we can divide all missing tuples into different groups such that all the tuples in the same group involve the same value of the new parameter. Doing so allows us to divide the vertical growth problem into multiple independent sub-problems, each of which tries to generate tests to cover one group of missing tuples. Note that, missing tuples in different groups must be covered with different tests. In this respect, there exists no interaction between missing tuples in different groups.

2.3.1 Example for 3-way test generation

We present an example to show why the vertical growth of IPOG needs to be improved for the *t*-way generation when *t* is greater than two: Assume that we are in the vertical growth phase of a 3-way test generation process. Let *d* be the new parameter being covered. Assume that the missing tuples involving a value of *d* denoted as *d*.1 are t1 = {*a*.2, *b*.0, *d*.1}, t2 = {*b*.1, *c*.0, *d*.1}, t3 = {*a*.2, *c*.0, *d*.1}, t4 = {*b*.0, *c*.1, *d*.1}. Figure 2-2 shows the conflict graph of this example; vertices are missing tuples and edges are conflicts. Two tuples have a conflict if and only if they cannot be covered by the same test. This happens when there exists at least one parameter that appears in both tuples but have different values in these two tuples.

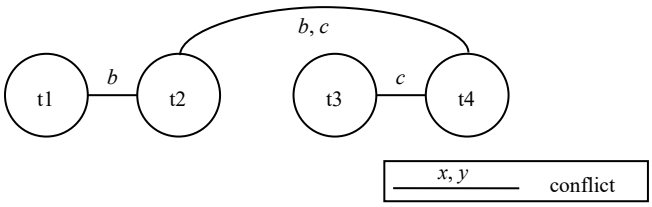

Figure 2-2. Conflict graph for the example 3-way test generation process

TABLE 2-1. A LESS OPTIMAL TEST SET FOR THE EXAMPLE 3-WAY TEST GENERATION PROCESS

Covered Tuples	**Test**			
	a	***b***	***c***	***d***
t1, t3	2	0	0	1
t2	*	1	0	1
t4	*	0	1	1

If we try to cover these missing tuples in a default order (t1, t2, t3, t4), we get 3 tests as shown in TABLE 2-1. ("*" represents "don't care" value.)

However, if the tuples are covered in the following order (t2, t4, t1, t3), only 2 tests are needed to cover all of them, as shown in TABLE 2-2.

TABLE 2-2. AN OPTIMAL TEST SET FOR THE EXAMPLE 3-WAY TEST GENERATION PROCESS

Covered Tuples	**Test**			
	a	***b***	***c***	***d***
t2, t3	2	1	0	1
t4, t1	2	0	1	1

The problem of how to use the minimum number of tests to cover a set of missing tuples can be modeled as the "Minimum Vertex Coloring (MVC)" problem in the conflict graph of these tuples.

A vertex coloring is an assignment of labels or colors to each vertex of a graph such that no edge connects two identically colored vertices. The MVC problem seeks to minimize the number of colors for a given graph. Such a coloring is referred to as a minimum vertex coloring, and the minimum number of colors with which the vertices of a graph G may be colored is called the chromatic number [10]. The MVC problem is a classical NP-hard optimization problem in computer science, and is typically solved using a greedy algorithm, e.g., greedy coloring [13, 14].

A greedy coloring colors of the vertices of a graph in a greedy manner. Specifically, it considers the vertices of the graph in sequence and assigns each vertex its first available color. Greedy colorings may not always result in the minimum number of colors. In particular, the order in which the vertices are covered has a significant impact on the result.

A commonly used ordering for greedy coloring is to choose a vertex of minimum degree, order the remaining vertices, and then place this vertex last in the ordering, which is equivalent to a non-increasing order of the degree of the vertices. If every sub-graph of a graph G contains a vertex of degree at most *d*, then the greedy coloring using this ordering will use at most *d+1* colors [15, 16].

2.3.2 Example for 2-way test generation

We use an example to show why the vertical growth of IPOG is optimal for pairwise: Assume that we are in the vertical growth phase of a 2-way test generation process. Let *d* be the new parameter being covered. Assume that the missing tuples involving a value of *d* denoted as *d*.1 are t1 = {*a*.0, *d*.1}, t2 = {*a*.1, *d*.1}, t3 = {*a*.2, *d*.1}, t4 = {*b*.0, *d*.1}, t5 = {*b*.1, *d*.1}, t6 = {*c*.1, *d*.1}. Figure 2-3 shows the conflict graph of this example that consists of three connected components, each of which is a complete graph with conflicts involving only one old parameter.

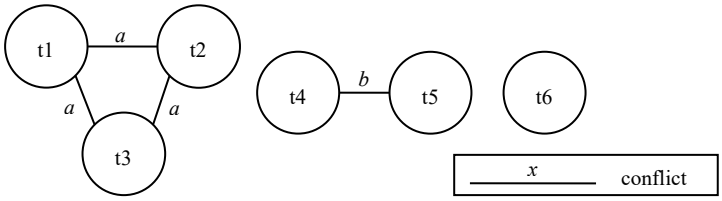

Figure 2-3. Conflict graph for the example 2-way test generation process

Any arbitrary order is optimal in the 2-way test generation, such as (t1, t2, t3, t4, t5, t6) or (t1, t4, t6, t2, t5, t3) whose test set size is 3. The reason can be presented as follows: For the largest complete sub-graph with degree *d*, it needs at least *d+1* colors to separately color its *d+1* vertices. While *d+1* colors are enough to color any complete sub-graph, these sub-graphs in the conflict

graph for 2-way test generation are all unconnected, *d+1* colors are enough to color the whole conflict graph. In this case, there is only one result for greedy coloring with any order, *d+1* colors, which is optimal. This explains why the original IPOG algorithm is optimal for 2-way test generation, even though it uses an arbitrary order to cover missing tuples during vertical growth.

2.4 Approach for IPOG Improvement

In Section 2.3, we only discussed the case where new tests are created to cover missing tuples, ignoring the case where existing tests resulted from horizontal growth can also be changed to cover missing tuples. If all the existing tests with "don't care" values are of type 1 (see Section 2.2) and thus cannot cover any more missing tuples, no additional action is needed; otherwise, it is important to properly represent the existing tests as colored vertices in the conflict graph with separate colors, and add edges to represent conflicts between missing tuples and existing tests. Note that in this case, the vertical growth phase of IPOG may no longer be optimal even for 2-way test generation, since the conflict graph including existing tests becomes more complex such that the arbitrary ordering of tuples does not necessarily produce the optimal result.

2.4.1 Conflict graph with existing tests

2.4.1.1 Graph model with type 2 existing tests

For an existing test in which only old parameters have the "don't care" value (called type 2 in Section 2.2), the new parameter is assigned a specific value. Thus, it can only cover missing tuples that have the same value for the new parameter. We refer to these missing tuples as missing

tuples relevant to this test. We represent the existing test as a colored vertex, and add edges to represent the conflicts between this test and missing tuples that are relevant to this test. Note that the colors of these colored vertices as existing tests are all different.

For example, let *f* be the new parameter being covered. Given two existing tests of type 2 as colored vertices T1 = {*a*.*, *b*.0, *c*.0, *d*.0, *e*.*, *f*.0}, T2 = {*a*.0, *b*.*, *c*.0, *d*.*, *e*.0, *f*.0}, the specific value of the new parameter is *f*.0. Assume that all the missing tuples involving *f*.0 are t1 = {*a*.0, *d*.0, *f*.0}, t2 = {*a*.0, *d*.1, *f*.0}, t3 = {*b*.0, *e*.0, *f*.0}, t4 = {*b*.0, *e*.1, *f*.0}, which are relevant to T1 and T2. Figure 2-4 shows the conflict graph. (Dash lined circle means this vertex has already been colored as an existing test.)

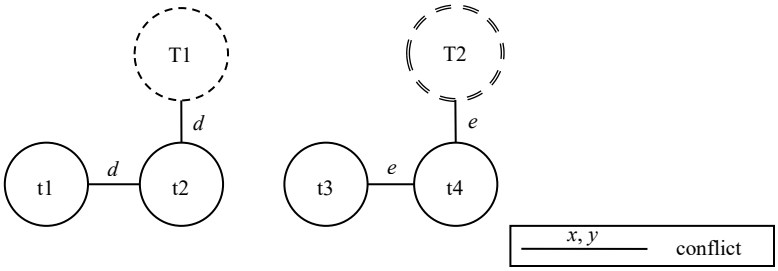

Figure 2-4. Conflict graph with two type 2 existing tests

TABLE 2-3. AN OPTIMAL TEST SET GENERATED FROM THE CONFLICT GRAPH IN FIG. 2-4

Existing Test	**Covered Tuples**	**Extended Test**					
		a	***b***	***c***	***d***	***e***	***f***
T1	t4, t1	0	0	0	0	1	0
T2	t2, t3	0	0	0	1	0	0

According to the conflict graph in Figure 2-4, we can get an optimal tuple order (t2, t4, t1, t3), in the order of non-increasing degrees. The degree of a tuple is the number of the conflicts it

involves. Following this order allows the tuples to be covered by the two existing tests, i.e., T1 and T2, without adding a new test, as shown in TABLE 2-3.

If we do not capture the conflicts between missing tuples and existing tests, i.e., the two conflicts between T1 and t2, T2 and t4, an insufficient conflict graph would be obtained as shown in Figure 2-5.

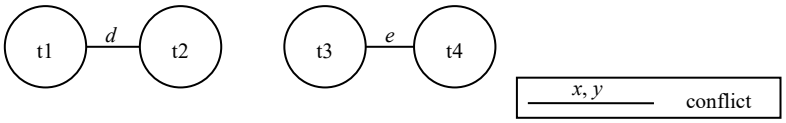

Figure 2-5. An insufficient conflict graph with two type 2 existing tests

If this insufficient conflict graph is used, the tuple order based on the non-increasing order of degrees would be (t1, t2, t3, t4), which would generate a third test, in addition to existing tests T1 and T2, in order to cover all these tuples, as shown in TABLE 2-4.

TABLE 2-4. A LESS OPTIMAL TEST SET FROM THE INSUFFICIENT CONFLICT GRAPH IN FIG. 2-5

Existing Test	**Covered Tuples**	**Extended Test**					
		a	***b***	***c***	***d***	***e***	***f***
T1	t1, t3	0	0	0	0	0	0
T2	t2	0	*	0	1	0	0
	t4	*	0	*	*	1	0

This example shows that it is important to capture the conflicts between existing tests and missing tuples.

2.4.1.2 Graph model with type 3 existing tests

For an existing test in which old parameters and the new parameter have the “don’t care” value, called type 3 in Section 2.2, the value of the new parameter can still be changed. A type 3

existing test provides opportunities for missing tuples involving any possible value of the new parameter to be covered, which leads to competition among missing tuples not only in the same group but also in different groups. Recall that missing tuples involving the same value of the new parameter are grouped together. This makes different groups no longer independent. In this respect, if there exists a type 3 test, missing tuples do not need to be grouped.

We represent missing tuples as uncolored vertices, and the existing tests as colored vertices with separate colors. Recall that two tuples have a conflict if they cannot be covered by the same test. One tuple and one existing test have a conflict if the tuple cannot be covered by changing the existing test. Note that, there may be conflicts due to different values of the new parameter since tuples are no longer grouped.

For example, let *f* be the new parameter being covered. Given a type 3 test as a colored vertex T1 = {*a*.0, *b*.0, *c*.0, *d*.*, *e*.*, *f*.*}, assume that all the missing tuples are t1 = {*c*.1, *d*.0, *f*.0}, t2 = {*d*.0, *e*.1, *f*.0}, t3 = {*d*.0, *e*.1, *f*.1}. Figure 2-6 shows the conflict graph.

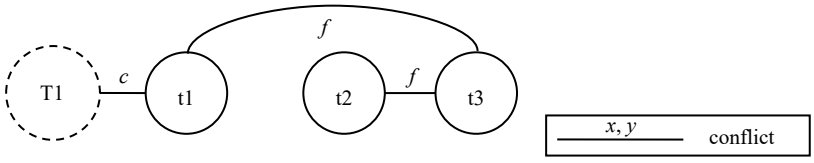

Figure 2-6. Conflict graph with a type 3 existing test

TABLE 2-5. AN OPTIMAL TEST SET GENERATED FROM THE CONFLICT GRAPH IN FIG. 2-6

Existing Test	**Covered Tuples**	**Extended Test**					
		a	***b***	***c***	***d***	***e***	***f***
T1	t3	0	0	0	0	1	1
	t1, t2	*	*	1	0	1	0

An optimal tuple order (t1, t3, t2) is derived from this conflict graph, which allows all the tuples to be covered by two tests, as shown in TABLE 2-5.

If we do not represent the conflicts between missing tuples involving different values of the new parameter *f*, i.e., the two conflicts between t1 and t3, t2 and t3, we would obtain an insufficient conflict graph as shown in Figure 2-7.

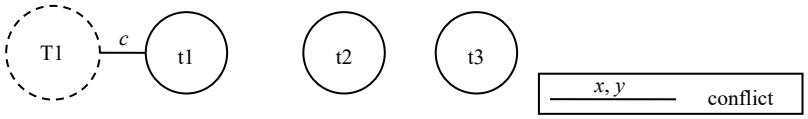

Figure 2-7. An insufficient conflict graph with a type 3 existing test

If this insufficient conflict graph is used, the tuple order based on the non-increasing order of degrees would be (t1, t2, t3), which would generate one more test as shown in TABLE 2-6.

TABLE 2-6. A LESS OPTIMAL TEST SET FROM THE INSUFFICIENT CONFLICT GRAPH IN FIG. 2-7

Existing Test	**Covered Tuples**	**Extended Test**					
		a	***b***	***c***	***d***	***e***	***f***
T1	t2	0	0	0	0	1	0
	t1	*	*	1	0	*	0
	t3	*	*	*	0	1	1

It indicates that, if there exist type 3 tests, the conflicts between missing tuples involving different values of the new parameter become important.

2.4.1.3 Graph model with type 2 and 3 existing tests

If there exist tests of type 2 and 3, the conflicts between missing tuples and type 2 existing tests involving different values of the new parameter can also be proved to be important.

For example, let *f* be the new parameter being covered. Given a type 3 test as a colored vertex T1 = {*a*.0, *b*.0, *c*.0, *d*.*, *e*.*, *f*.*}, and a type 2 test as a colored vertex T2 = {*a*.1, *b*.0, *c*.0, *d*.*, *e*.*, *f*.0}, assume that all the missing tuples are t1 = {*c*.0, *d*.0, *f*.0}, t2 = {*d*.0, *e*.1, *f*.1}. Figure 2-8 shows the conflict graph.

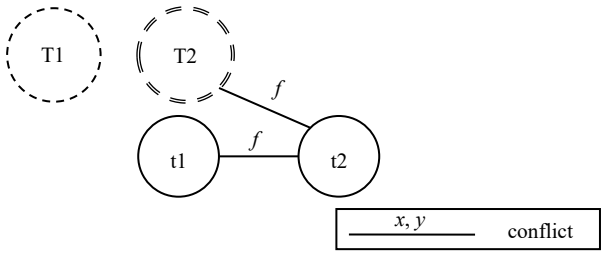

Figure 2-8. Conflict graph with a type 3 existing test and a type 2 test

An optimal tuple order (t2, t1) is derived from this conflict graph, which allows all the tuples to be covered by two tests, as shown in TABLE 2-7.

TABLE 2-7. AN OPTIMAL TEST SET GENERATED FROM THE CONFLICT GRAPH IN FIG. 2-8

Existing Test	**Covered Tuples**	**Extended Test**					
		a	***b***	***c***	***d***	***e***	***f***
T1	t2	0	0	0	0	1	1
T2	t1	1	0	0	0	*	0

If we do not represent the conflict between the missing tuple and the type 2 test involving different values of the new parameter *f*, i.e., the conflict between t2 and T2, we would obtain an insufficient conflict graph as shown in Figure 2-9.

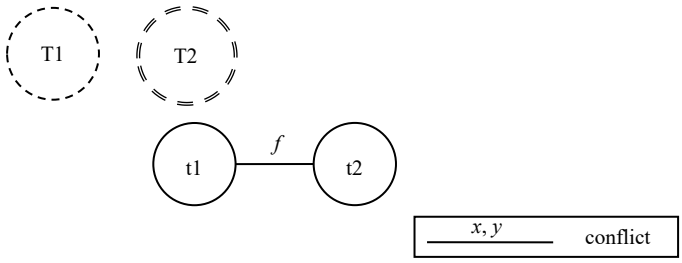

Figure 2-9. An insufficient conflict graph with a type 3 test and a type 2 test

If this insufficient conflict graph is used, the tuple order based on the non-increasing order of degrees would be (t1, t2), which would generate one more test as shown in TABLE 2-8.

TABLE 2-8. A LESS OPTIMAL TEST SET FROM THE INSUFFICIENT CONFLICT GRAPH IN FIG. 2-9

Existing Test	**Covered Tuples**	**Extended Test**					
		a	***b***	***c***	***d***	***e***	***f***
T1	t1	0	0	0	0	*	0
T2		1	0	0	*	*	0
	t2	*	*	0	*	1	1

This scenario only happens when a type 3 test is in front of a type 2 test, which not often occurs during the horizontal growth phase of IPOG, but still there is a chance.

In summary, if there are existing tests of type 2 or 3, we must capture conflicts between missing tuples and existing tests in the conflict graph. Also, there may exist more conflicts for tests of type 3, as missing tuples cannot be grouped in this case.

2.4.2 Implementation

The vertical growth phase (Figure 2-1 lines 12-19) of our original IPOG is already a greedy algorithm that covers each tuple in the first test encountered that could cover this tuple, which is

similar to greedy coloring. So, the work needed to implement the proposed improvement is to build the conflict graph for missing tuples and existing tests, and then to derive the tuple order from the conflict graph. Note that once a tuple is covered in a test, it may also cover other missing tuples which should be marked as covered.

First, we use a matrix to represent the conflict graph consisting of missing tuples, existing tests and their conflicts. For instance, assume that the new parameter is *d* and there are missing tuples t1 = {*a*.0, *b*.1, *d*.0}, t2 = {*b*.1, *c*.1, *d*.0}, t3 = {*a*.0, *b*.1, *d*.1}, t4 = {*a*.1, *c*.1, *d*.1} and existing tests T1 = {*a*.1, *b*.0, *c*.*, *d*.0}, T2 = {*a*.0, *b*.*, *c*.0, *d*.1}, T3 = {*a*.1, *b*.1, *c*.1, *d*.*}. We create a conflict matrix, which consists of an (upper triangular) matrix that capture conflicts between missing tuples, and a matrix that capture conflicts between missing tuples and existing tuples, shown as TABLE 2-9.

TABLE 2-9. AN EXAMPLE CONFLICT MATRIX

		***d*.0**		***d*.1**		***d*.0**	***d*.1**	***d*.***
		t1	**t2**	**t3**	**t4**	**T1**	**T2**	**T3**
***d*.0**	**t1**	F	F	T	T	T	T	T
	t2		F	T	T	T	T	F
***d*.1**	**t3**			F	T	T	F	T
	t4				F	T	T	F

In the above conflict matrix, only the values in the sub-matrices in bold-lined boxes need to be determined by comparing the values of the parameters that appear in both tuples indicated by the row and column indices. All other values in the rest of the matrix must be true, since combinations with different values of the new parameter always conflict with each other. We use

this observation to reduce the space requirement for storing conflict matrix. The reduction can be significant when the number of missing tuples *m* is large and/or domain size *s* is big.

According to the above conflict matrix, we can easily find the degree of each tuple by counting the conflict flags in its involved row and column, such as degree(t1) = 5, degree(t2) = 4, degree(t3) = 5, degree(t4) = 5.

TABLE 2-10. AN OPTIMAL TEST SET GENERATED FROM AN OPTIMAL ORDER

Existing Test	**Covered Tuples**	**Extended Test**			
		a	***b***	***c***	***d***
T1		1	0	*	0
T2	t3	0	1	0	1
T3	t4	1	1	1	1
	t1, t2	0	1	1	0

The order of tuples to be covered can be determined as follows. We first choose a missing tuple of maximum degree in the conflict graph, and then cover it in its first compatible test by searching in the extending test set from top to bottom. In this example, the order of tuples is determined to be (t1, t3, t4, t2), which can be covered by four tests as shown in TABLE 2-10.

2.5 Experiment

We implemented in ACTS a revised IPOG algorithm that incorporates the Graph Coloring-based (GC) approach to vertical growth. We report several experiments that were conducted to evaluate the effectiveness of the revised IPOG algorithm. In particular, we conducted an experiment that compared ACTS with an existing tool PICT [8]. The experimental

results show that the revised IPOG algorithm performed better than the original IPOG algorithm in ACTS and PICT for a set of real-life systems.

2.5.1 Experiment design

TABLE 2-11 shows nine real-life systems used in our experiments. These systems have been packaged as example SUTs in ACTS releases for several years.

TABLE 2-11. MODELS AND SOURCES

Software Name	SUT File Name	Source
Apache HTTP server	apache.xml	http://httpd.apache.org
Berkeley DB	Berkeley.xml	https://oss.oracle.com/berkeley-db.html
Bugzilla bug tracker	bugzilla.xml	https://bugzilla.mozilla.org
GCC compiler	gcc.xml	https://gcc.gnu.org
replace c program	replace.xml	SIR repository, http://sir.unl.edu
SPIN simulator	Spin_S.xml	http://spinroot.com
SPIN verifier	Spin_V.xml	http://spinroot.com
tcas c program	tcas.xml	SIR repository, http://sir.unl.edu
Violet UML editor	Violet.xml	http://sourceforge.net/projects/violet

We adopt the exponential notation to denote parameter configurations, where d^n means that there are n parameters of domain size d. The configurations of these real-life systems are shown in TABLE 2-12. (Recall that we do not handle constraints in this paper.)

TABLE 2-12. CONFIGURATIONS OF REAL-LIFE SYSTEMS

Name	Num. of Parameters	Parameter Configuration
apache	172	$2^{158}\ 3^8\ 4^4\ 5^1\ 6^1$
Berkeley	78	2^{78}
bugzilla	52	$2^{49}3^14^2$
gcc	199	$2^{189}3^{10}$

Name	Num. of Parameters	Parameter Configuration
replace	20	$2^4 4^{16}$
Spin_S	18	$2^{13} 4^5$
Spin_V	55	$2^{42} 3^2 4^{11}$
tcas	12	$2^7\ 3^2\ 4^1\ 10^2$
Violet	101	2^{101}

2.5.2 Results and analysis

The experimental environment is set up as the following: OS: Windows 7 64bits, CPU: Intel Dual-Core i5 2.5GHz, Memory: 4 GB DDR3, Java SDK (Since ACTS is a Java tool): Java SE 1.6, Java Max Heap Size: 1024MB.

We generate 4-way, 3-way and 2-way test sets for the nine subject systems, by using PICT 3.3, ACTS 2.92, and ACTS 2.92-GC. Experimental results are shown in TABLE 2-13, TABLE 2-14 and TABLE 2-15. Note that cells highlighted by gray background indicate the smallest test set sizes produced by the three tools.

TABLE 2-13. RESULT OF 4-WAY TEST GENERATION

Name	**# of Tuples**	**PICT 3.3**		**ACTS 2.92 original IPOG**		**ACTS 2.92-GC revised IPOG**	
		size	***Time(s)***	***size***	***Time(s)***	***size***	***Time(s)***
apache	728304446	N/A	Crash	834	1456.944	828	1523.073
Berkeley	22822800	121	188.168	120	20.178	119	20.312
bugzilla	5204192	220	64.609	230	3.886	227	4.147
gcc	N/A	N/A	Crash	N/A	OOM	N/A	OOM
replace	800784	1062	23.380	987	0.497	970	62.263
Spin_S	125040	353	1.364	343	0.079	341	0.137
Spin_V	11873396	808	430.535	779	12.336	749	57.495
tcas	64696	1410	2.274	1359	0.070	1358	0.443
Violet	65326800	131	588.096	131	67.896	132	71.572

OOM = Out of Memory Java exception

TABLE 2-13 shows that for 4-way test generation, compared with the original IPOG implementation, the revised IPOG algorithm reduces test set size for seven of the nine real-life systems, and slightly increases test set size for one of the nine systems. Note that for gcc, none of the three tools can generate 4-way test sets due to limited memory space.

TABLE 2-14. RESULT OF 3-WAY TEST GENERATION

Name	**# of Tuples**	**PICT 3.3**		**ACTS 2.92 original IPOG**		**ACTS 2.92-GC revised IPOG**	
		size	***Time(s)***	***size***	***Time(s)***	***size***	***Time(s)***
apache	8087048	202	132.912	173	10.007	170	10.315
Berkeley	608608	45	1.831	46	0.320	44	0.543
bugzilla	203104	68	0.648	67	0.131	66	0.210
gcc	11147562	88	89.99	78	17.238	76	18.582
replace	52768	203	0.343	181	0.058	181	0.580
Spin_S	13328	96	0.071	79	0.036	80	0.094
Spin_V	377128	168	2.300	159	0.227	156	0.758
tcas	9158	402	0.124	400	0.045	400	0.038
Violet	1333200	47	3.770	48	0.826	48	1.114

TABLE 2-14 shows that for 3-way test generation, compared with the original IPOG algorithm, the revised IPOG algorithm reduces test set size for five of the nine real-life systems, and slightly increases test set size for one of the nine systems. Both algorithms produce the same test set size for the other three systems.

These results show that our new vertical growth algorithm is effective for *t*-way test generation when *t* is greater than two.

TABLE 2-15. RESULT OF 2-WAY TEST GENERATION

Name	**# of Tuples**	**PICT 3.3**		**ACTS 2.92 original IPOG**		**ACTS 2.92-GC revised IPOG**	
		size	***Time(s)***	***size***	***Time(s)***	***size***	***Time(s)***
apache	66930	39	0.204	33	0.112	33	0.123
Berkeley	12012	15	0.052	16	0.055	16	0.077
bugzilla	5822	20	0.035	18	0.038	18	0.044
gcc	82809	21	0.133	20	0.148	20	0.173
replace	2456	38	0.044	38	0.026	37	0.061
Spin_S	992	23	0.033	24	0.025	24	0.030
Spin_V	8797	32	0.047	33	0.046	34	0.067
tcas	837	100	0.038	100	0.037	100	0.032
Violet	20200	16	0.053	16	0.080	16	0.095

TABLE 2-15 shows that for 2-way test generation, compared with the original IPOG algorithm, the revised IPOG algorithm only slightly reduces test set size for one of the nine real-life systems, and slightly increases test set size for one of the nine systems. Both algorithms produce the same test set size for the other seven systems. This is expected, as the vertical growth phase in the original IPOG algorithm is optimal for 2-way testing.

2.6 Related Work

In our past work, we presented three variants of the IPO algorithm: IPOG [3], IPOG-F/IPOG-F2 [17] and IPOG-D [4]. IPOG implements the generalization to *t*-way testing of the original IPO algorithm. This algorithm explicitly enumerates all possible combinations, and does not scale well to big systems, where the number of combinations is large, or when resources are limited. IPOG-F is a variant of IPOG whose implementation has been optimized for speed, which also achieves better test set size than IPOG for some systems, but gets worse for some

other systems [18]. IPOG-D is a variant of IPOG, incorporating a recursive technique, namely the doubling-construct, and developed just to address the problem of reducing the number of combinations that have to be enumerated, so to improve the algorithm scalability.

Younis et al. [19-21] presented MIPOG which is a modification of the IPOG algorithm for *t*-way testing. The differences between the MIPOG and IPOG algorithms lie in both horizontal and vertical extensions. In horizontal extension, the MIPOG algorithm checks all the values of the input parameter and chooses the value that contains the maximum number of uncovered tuples. Also, it optimizes "don't care" values. In vertical extension, MIPOG reorders the set of missing tuples in the decreasing order of the size of the remaining tuples. After that, it chooses the first tuple from the missing set and combines that tuple with others missing tuples (i.e., the resulting test case must have the maximum weight of the uncovered tuples). Their results show that the MIPOG algorithm will always give the same or less test set than IPOG. MIPOG uses a different approach than our work. Also, it does not provide a public tool, which prevents an experimental comparison between the two approaches.

2.7 Conclusion and Future Work

In this paper, we presented an improvement on the vertical growth phase of the IPOG algorithm. In this algorithm we create a graph model that captures the conflict relationship among different tuples and existing tests. We reduce the vertical growth problem to a classical NP-hard problem "Minimum Vertex Coloring" on this graph. A greedy coloring algorithm is used to

determine the order in which missing tuples can be covered. Specifically, the higher the degree of a missing tuple in the conflict graph, the earlier it should be covered. The experimental results show that the improvement can further reduce the number of tests that are generated by the IPOG algorithm for a set of real-life systems.

In the future, we will consider the impact of constraints on vertical growth. Constraints will create more conflicts between tuples. In addition, constraints may introduce conflicts that involve more than two tuples. We will explore how to represent such higher-degree conflicts and how to consider them during vertical growth.

2.8 Acknowledgment

This work is partly supported by a research grant (70NANB12H175) from Information Technology Laboratory of National Institute of Standards and Technology (NIST).

DISCLAIMER: NIST does not endorse or recommend any commercial product referenced in this paper or imply that a referenced product is necessarily the best available for the purpose.

2.9 References

[1] Y. Lei, and K-C. Tai. “In-parameter-order: A test generation strategy for pairwise testing.” In High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International, pp. 254-261. IEEE, 1998.

[2] K-C Tai, and Y. Lei. “A test generation strategy for pairwise testing.” IEEE Transactions on Software Engineering 28, no. 1 (2002): 109-111.

[3] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. “IPOG: A general strategy for t-way software testing.” In Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the, pp. 549-556. IEEE, 2007.

[4] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, and J. Lawrence. “IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing.” Software Testing, Verification and Reliability 18, no. 3 (2008): 125-148.

[5] L. Yu, Y. Lei, M. Nourozborazjany, R.N. Kacker, and D. R. Kuhn. “An efficient algorithm for constraint handling in combinatorial test generation.” In Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, pp. 242-251. IEEE, 2013.

[6] ACTS (Advanced Combinatorial Testing System), http://csrc.nist.gov/acts/

[7] L. Yu, Y. Lei, R.N. Kacker, and D.R. Kuhn. “Acts: A combinatorial test generation tool.” In Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, pp. 370-375. IEEE, 2013.

[8] PICT (Pairwise Independent Combinatorial Testing tool), http://blogs.msdn.com/b/nagasatish/archive/2006/11/30/pairwise-testing-pict-tool.aspx

[9] J. Czerwonka. “Pairwise testing in the real world: Practical extensions to test-case scenarios.” In Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer, pp. 419-430. 2006.

[10] N. Christofides. “An algorithm for the chromatic number of a graph.” The Computer Journal 14, no. 1 (1971): 38-39.

[11] D.W. Matula, G. Marble, and J.D. Isaacson. “Graph coloring algorithms.” Graph theory and computing (1972): 109-122.

[12] S. Skiena. “Finding a Vertex Coloring.” §5.5.3 in Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Reading, MA: Addison-Wesley, pp. 214-215, 1990.

[13] B. Manvel. “Extremely greedy coloring algorithms.” Graphs and Applications (1985): 257-270.

[14] L. Kučera. “The greedy coloring is a bad probabilistic algorithm.”d Journal of Algorithms 12, no. 4 (1991): 674-684.

[15] D.J. Welsh, and M.B. Powell. “An upper bound for the chromatic number of a graph and its application to timetabling problems.” The Computer Journal 10, no. 1 (1967): 85-86.

[16] M.M. Sysło. “Sequential coloring versus Welsh-Powell bound.” Annals of Discrete Mathematics 39 (1989): 241-243.

[17] M. Forbes, J. Lawrence, Y. Lei, R.N. Kacker, and D.R. Kuhn. “Refining the in-parameter-order strategy for constructing covering arrays.” Journal of Research of the National Institute of Standards and Technology 113, no. 5 (2008): 287-297.

[18] A. Calvagna, and A. Gargantini. “T-wise combinatorial interaction test suites construction based on coverage inheritance.” Software Testing, Verification and Reliability 22, no. 7 (2012): 507-526.

[19] M.I. Younis, K.Z. Zamli, and N.M. Isa. “MIPOG-Modification of the IPOG strategy for T-Way software testing.” Proceeding of the Distributed Frameworks and Applications (DFmA) (2008).

[20] M.I. Younis, K.Z. Zamli, and N.A.M. Isa. “A strategy for grid based t-way test data generation.” In Distributed Framework and Applications, 2008. DFmA 2008. First International Conference on, pp. 73-78. IEEE, 2008.

[21] M.I. Younis, and K.Z. Zamli. “MIPOG-An Efficient t-Way Minimization Strategy for Combinatorial Testing.” Int. J. Comput. Theory Eng 3, no. 3 (2011): 388-397.

OPTIMIZING IPOG'S VERTICAL GROWTH WITH CONSTRAINTS BASED ON HYPERGRAPH COLORING

Feng Duan, Yu Lei, Linbin Yu, Raghu N. Kacker, and D. Richard Kuhn. “Optimizing IPOG's Vertical Growth with Constraints Based on Hypergraph Coloring.” In 10th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 181-188. IEEE, 2017.

CHAPTER 3.

Optimizing IPOG's Vertical Growth with Constraints Based on Hypergraph Coloring

Abstract—In this paper, we present an optimization of IPOG's vertical growth phase in the presence of constraints. The vertical growth problem is modeled as a classical NP-hard graph problem called "Minimum Vertex Coloring". In the graph model, vertices are either missing tuples that are waiting to be colored or existing tests that are already colored in distinct colors at the initial state; edges/hyperedges are conflicts among vertices that cannot be put in a same test. After coloring, a group of vertices in same color can be transformed to exactly a valid test. Since the original IPOG algorithm uses an arbitrary order to cover missing tuples during vertical growth, in order to reduce the number of tests, we compute the Degree of Conflicts (DOC) for each tuple, and cover the tuples in the non-increasing order of DOC. We implement a new IPOG algorithm incorporating this optimization. The experimental results show that the new IPOG algorithm reduces the number of tests for many real-life systems with constraints.

Keywords—*Combinatorial testing; Multi-way test generation; ACTS; Hyperedge; Minimum vertex coloring; Tuple ordering; Constraint handling; Minimum Forbidden Tuples*

3.1 Introduction

In our earlier work, we developed a *t*-way test generation algorithm called In-Parameter-Order-General (IPOG) [3][4][5]. The IPOG algorithm is a generalization of a 2-way

(a.k.a. pairwise) test generation algorithm called In-Parameter-Order (IPO) [1][2]. For a system with *t* or more parameters, the IPOG algorithm builds a *t*-way test set for the first *t* parameters, extends the test set to build a *t*-way test set for the first *t+1* parameters, and then continues to extend the test set until it builds a *t*-way test set for all the parameters.

The extension of an existing *t*-way test set for a new parameter is performed in two phases:

- Horizontal growth, which extends each existing test by adding one value for the new parameter;
- Vertical growth, which adds, if needed, new tests to cover missing tuples, i.e., tuples that have not been covered yet.

The challenge of vertical growth is to minimize the number of new tests. The vertical growth part of the original IPO algorithm is optimal for 2-way test generation without constraints. When the IPO algorithm is extended for general t-way testing, the vertical growth part is extended in a straightforward manner. The vertical growth part of the IPOG algorithm is not optimal for general t-way test generation with or without constraints.

In our earlier work [15], we reduced the vertical growth problem to a classical graph coloring problem called "Minimum Vertex Coloring" [10][11]. We represented missing tuples as uncolored vertices, and the existing tests as colored vertices with distinct colors. Based on this reduction, we developed an algorithm that improves the vertical growth part of the IPOG algorithm. However, in [15], we only considered compatibility conflicts, i.e., conflicts between

two missing tuples, or between one existing test and one missing tuple, that assign different values on the same parameter. We did not consider validity conflicts, i.e., conflicts that are due to constraints. A validity conflict is difficult to handle in a classical graph coloring problem as it may involve more than two vertices and thus cannot be represented as an edge in a graph.

In this paper, we extend our earlier work to handle validity conflicts. In particular, we use the notion of hyperedge to represent validity conflicts and reduce the vertical growth problem to a hypergraph coloring problem [14]. A hyperedge is an edge that may involve more than two vertices.

The key challenge of our approach is to create hyperedges to represent validity conflicts from constraints. In our approach, we first transform constraints into Minimum Forbidden Tuples (MFTs) [17][18][19]. A forbidden tuple is a tuple that violates constraints. A minimum forbidden tuple is a forbidden tuple of minimum size that covers no other forbidden tuples. For each MFT, we enumerate combinations of vertices that cover all parameter values of the MFT, and represent them as hyperedge candidates. Hyperedge candidates are formed by at least two missing tuples, or one existing test and at least one missing tuple. Finally, if a candidate is compatible, i.e., it doesn't contain compatibility conflict, and minimum, i.e., it doesn't contain smaller-size hyperedge, it is a hyperedge that represents a validity conflict.

The edges as compatibility conflicts and the hyperedges as validity conflicts are then used to calculate the Degree of Conflicts (DOC) for each uncolored vertex which may be involved in

these edges/hyperedges. And a greedy hypergraph coloring method is deployed to cover the uncolored vertices in the non-increasing order of DOC, i.e., cover tuples which involve more conflicts earlier.

We implemented a new version of the IPOG algorithm that incorporates the hypergraph coloring-based optimization in ACTS [6][7]. We conducted experiments on a set of real-life systems collected from the literature. These systems have been used to evaluate the effectiveness of *t*-way test generation algorithms. The experimental results show that the new IPOG algorithm performed better than the original IPOG algorithm implemented in ACTS. For example, the new algorithm reduced test set size for 4 of 9 systems in 2-way and for 6 of 9 systems in 3-way, and better than PICT [8][9], for these real-life systems.

The remainder of the paper is organized as follows. Section 3.2 briefly reviews the vertical growth of IPOG algorithm with constraint handling. It also shows that vertical growth is exactly a concrete greedy coloring algorithm. Section 3.3 describes two motivating examples to show why the vertical growth phase of IPOG is not optimal for test generation with constraints. Section 3.4 presents a reduction of the vertical growth problem to a hypergraph coloring problem, and the optimization of vertical growth based on the hypergraph coloring problem. Section 3.5 reports the design and the results of the experiments. Section 3.6 discusses related work on graph methods for test generation, IPOG vertical growth optimizations, and some constraint handling methods. Section 3.7 provides concluding remarks and our plan for future work.

3.2 Vertical Growth vs Greedy Coloring

In this section, we present the vertical growth part of IPOG algorithm with constraint handling, as shown in Figure 3-1. For more details, please refer to our earlier paper [5].

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 target tuples involving parameter *p* and any group of (*t-1*) parameters among the first *k* parameters. These tuples 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 tuples. During vertical growth, the remaining tuples 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 tuple, parameters that are not involved in the tuple are given a special value called "don't care". These "don't care" values can be later changed to cover other tuples.

Algorithm *IPOG-Vertical-Growth*
Input
ts : existing test set
π : the set of *t*-way missing tuples involving new parameter *p*
c : constraints (MFTs)
Output
ts : updated test set
{
1. **for** (each tuple σ in set π){
2. **if** (there exists a test τ in test set *ts* that can be changed to a test τ' that covers both τ and σ , i.e., τ is **compatible** with σ, and τ' is **valid** on *c*) {
3. replace test τ with τ' in *ts*
4. remove from π the tuples covered by τ'
5. } **else** {
6. add a new test τ only contains σ into *ts*
7. remove from π the tuples covered by τ

```
8.        } // end if at line 2
9.     } // end for at line 1
10. return ts
}
```

Figure 3-1. The vertical growth step of the original IPOG with constraint handling

The vertical growth phase of IPOG is a tuple-oriented algorithm, i.e., to cover each missing tuple in an arbitrary order by filling it into an existing test or creating a new test. If we consider missing tuples as uncolored vertices, and existing tests as colored vertices with distinct colors, then vertical growth is equivalent to a greedy algorithm for vertex coloring.

Briefly, the equivalence of vertical growth and greedy coloring [12][13] works as follows: It sequentially picks up a missing tuple (uncolored vertex) from set π, tries to find the first available test (color) to fill the missing tuple into it (i.e., color the uncolored vertex with an existing color). A test being available w.r.t. a missing tuple means there exists no compatibility and validity conflicts between the test and the tuple (i.e., the colored vertices in the same color with the uncolored vertex involve no conflict). If none of the existing tests is available, a new test is created (i.e., color the uncolored vertex with a new color).

After coloring, each group of vertices in the same color is equivalent to a final test. If a group includes an existing test vertex, the final test is created by filling in the corresponding existing test with the missing tuples represented by the other (tuple) vertices. Otherwise, a new test is created by merging all the missing tuples represented by these vertices. Thus, the problem of how to

minimize the number of new tests can be reduced to the problem of how to minimize the number of colors in the hypergraph coloring problem.

3.3 Motivating Examples

Assume that the horizontal growth phase of IPOG has been finished for a given system, and the vertical growth phase is about to begin. Since the new parameter is involved in every missing tuple, we can divide all missing tuples into different groups such that all the tuples in the same group involve the same value of the new parameter. Doing so allows us to divide the vertical growth problem into multiple independent sub-problems, each of which tries to generate tests to cover one group of missing tuples. Note that missing tuples in different groups must be covered with different tests. In this respect, there exists no interaction between missing tuples in different groups.

3.3.1 Example for 3-way test generation

We present an example to show why the vertical growth of IPOG can be improved for the *t*-way generation, especially when *t* is greater than two. Assume that we are in the vertical growth phase of a 3-way test generation process. Let *f* be the new parameter being covered, and *f*.1 be the second value of *f*. Assume that the missing tuples involving *f*.1 are t1 = {*a*.1, *b*.0, *f*.1}, t2 = {*b*.1, *c*.0, *f*.1}, t3 = {*a*.1, *c*.0, *f*.1}, t4 = {*b*.0, *c*.1, *f*.1}, t5 = {*a*.1, *d*.0, *f*.1}, t6 = {*a*.1, *e*.0, *f*.1} as shown in TABLE 3-1.

TABLE 3-1. A MOTIVATING EXAMPLE OF 3-WAY MISSING TUPLES

Missing Tuples	**Parameter Values**					
	a	***b***	***c***	***d***	***e***	***f***
t1	1	0				1
t2		1	0			1
t3	1		0			1
t4		0	1			1
t5	1			0		1
t6	1				0	1

We assume that there are two constraints in the MFT format: MFT1 = {*b*.1, *d*.0, *e*.0}, MFT2 = {*c*.1, *e*.0}. MFT1 makes the combination of tuples {t2, t5, t6} become invalid, since {t2, t5, t6} = {*a*.1, *b*.1, *c*.0, *d*.0, *e*.0, *f*.1} contains {*b*.1, *d*.0, *e*.0}. Similarly, MFT2 makes the combination of tuples {t4, t6} become invalid, since {t4, t6} = {*a*.1, *b*.0, *c*.1, *e*.0, *f*.1} contains {*c*.1, *e*.0}.

Figure 3-2 shows the conflict hypergraph of this example, where vertices are missing tuples and edges are conflicts. Multiple tuples (no less than two) have a conflict if and only if they cannot be covered by the same test. There are two kinds of conflicts: compatibility conflict and validity conflict. Two tuples have a compatibility conflict if and only if there exists at least one parameter that appears in both tuples but has different values in these two tuples. Multiple tuples have a validity conflict if and only if they are compatible but invalid in the same test. This happens when the combination of these tuples violates constraints, i.e., it contains at least one MFT.

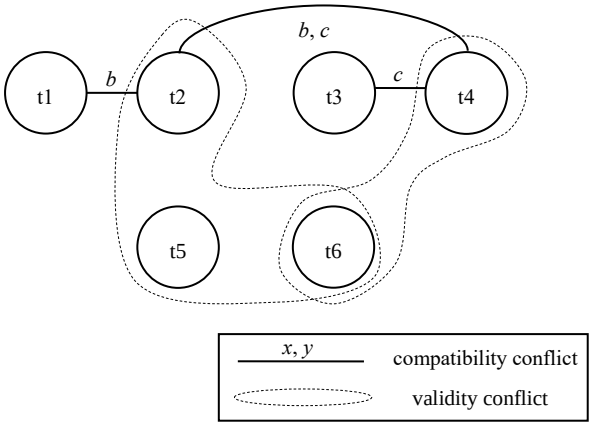

Figure 3-2. Conflict hypergraph for the example 3-way test generation process

In Figure 3-2, although we represent compatibility conflicts as simple edges and represent validity conflicts in hyperedge form for easy to read, both kinds of conflicts are edges of the hypergraph mathematically.

If we try to cover these missing tuples in a default order (t1, t2, t3, t4, t5, t6), we get 3 tests as shown in TABLE 3-2. ("*" represents "don't care" value.)

TABLE 3-2. A LESS OPTIMAL TEST SET FOR THE EXAMPLE 3-WAY TEST GENERATION PROCESS

Covered Tuples	**Test**					
	a	***b***	***c***	***d***	***e***	***f***
t1, t3, t5, t6	1	0	0	0	0	1
t2	*	1	0	*	*	1
t4	*	0	1	*	*	1

Our previous Graph Coloring-based approach [15] only counts the degree of compatibility conflicts for each vertex. Thus, degree(t1) = 1, degree(t2) = 2, degree(t3) = 1, degree(t4) = 2, degree(t5) = 0, degree(t6) = 0. Using these degrees, we obtain the GC order as (t2, t4, t1, t3, t5, t6), which results in 3 tests as shown in TABLE 3-3.

TABLE 3-3. AN GC-BASED TEST SET FOR THE EXAMPLE 3-WAY TEST GENERATION PROCESS

Covered Tuples	**Test**					
	a	***b***	***c***	***d***	***e***	***f***
t2, t3, t5	1	1	0	0	*	1
t4, t1	1	0	1	*	*	1
t6	1	*	*	*	0	1

However, if the tuples are covered in the following order (t4, t2, t6, t1, t3, t5), only 2 tests are needed to cover all of them, as shown in TABLE 3-4.

TABLE 3-4. AN OPTIMAL TEST SET FOR THE EXAMPLE 3-WAY TEST GENERATION PROCESS

Covered Tuples	**Test**					
	a	***b***	***c***	***d***	***e***	***f***
t4, t1, t5	1	0	1	0	*	1
t2, t6, t3	1	1	0	*	0	1

The intuition is that we should cover tuples which involve more conflicts earlier, since they are more strictly restrained by other missing tuples, existing tests, constraints, and thus have less freedom in terms that fewer tests can be used to cover these tuples.

The calculation on compatibility conflicts is quite easy, since they are all simple edges, which we have already discussed in previous paper [15]. In this paper, we mainly focus on how to extract hyperedges as validity conflicts from constraints, and how to use them to determine the coloring order since hyperedges may be of different sizes. The size of a hyperedge is the number of vertices involved in the hyperedge.

3.3.2 Example for 2-way test generation

Unlike the IPOG algorithm without constraints, which is optimal for 2-way testing as shown in our previous paper [15], we use an example to show that with constraints, even for 2-way testing, the vertical growth of IPOG is not optimal. Assume that we are in the vertical growth phase of a 2-way test generation process. Let *d* be the new parameter being covered, and *d*.1 be the second value of *d*. Assume that the missing tuples involving *d*.1 are t1 = {*a*.0, *d*.1}, t2 = {*a*.1, *d*.1}, t3 = {*b*.0, *d*.1}, t4 = {*b*.1, *d*.1}, t5 = {*c*.0, *d*.1}, t6 = {*c*.1, *d*.1}, and one constraint as MFT1 = {*a*.1, *b*.1, *c*.1} which makes the combination of tuples {t2, t4, t6} become invalid. Figure 3-3 shows the conflict hypergraph of this example.

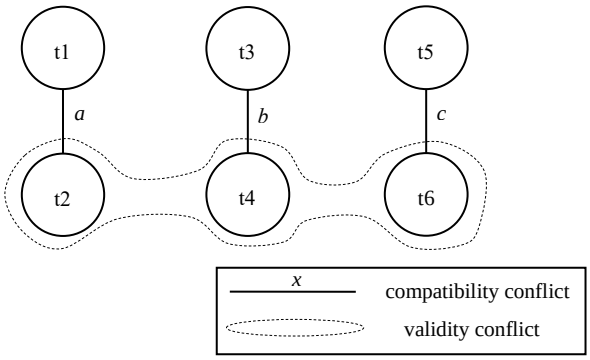

Figure 3-3. Conflict hypergraph for the example 2-way test generation process

If we try to cover these missing tuples in a default order (t1, t2, t3, t4, t5, t6) which is the same as the GC order, we get 3 tests: {t1, t3, t5}, {t2, t4}, {t6}.

However, if the tuples are covered in the following order (t2, t4, t6, t1, t3, t5), we would get an optimal test set having only 2 tests: {t2, t4, t5}, {t6, t1, t3}.

This example shows that, in order to minimize the test set, validity conflicts from constraints should be carefully considered even for 2-way test generation.

3.4 Hypergraph Coloring-based optimization

3.4.1 Reduction to Hypergraph Coloring

In mathematics, a hypergraph is a generalization of a graph in which an edge can join any number of vertices (also called hyperedge). A vertex coloring of a graph is a labeling of the graph's vertices with colors such that no two vertices sharing the same edge have the same color. Hypergraph coloring [14] is assigning one of the colors from a color set to every vertex of a hypergraph in such a way that each hyperedge contains at least two vertices of distinct colors. In other words, all vertices in a hyperedge cannot have only one color if the number of its vertices is no less than two. In this sense, hypergraph coloring is a direct generalization of graph coloring.

As mentioned in previous sections, vertical growth problem can be reduced to a greedy coloring problem on a hypergraph such that: missing tuples are mapped to uncolored vertices; existing tests are mapped to colored vertices with distinct colors; compatibility conflicts are mapped to edges; validity conflicts are mapped to hyperedges. After coloring, each group of vertices in the same color is considered as a test.

Hypergraph reduction is to create a hypergraph using the inputs of vertical growth. The inputs include: the set of constraints as Minimum Forbidden Tuples (MFTs) [17][18][19], the new parameter Pn being covered, the existing test set after horizontal growth, and the missing tuples.

As in section 3.3, we grouped missing tuples based on their new parameter values, in order to divide the vertical growth problem into multiple independent sub-problems. However, we cannot ignore the interaction between missing tuples and existing tests in same group. We also cannot ignore the effect of some existing tests, which would be contested by multiple groups of missing tuples. In this section, these aspects will be discussed.

Here is an example of hypergraph reduction: Given a system having parameters *a, b, c, d, e, f* ... with domain [0, 1], and an MFT as MFT1 = {*a*.1, *b*.0, *d*.0, *e*.0, *f*.1}. Let *f* be the new parameter being covered. Assume that, after horizontal growth, there are existing tests as T1 = {*a*.1, *b*.0, c.*, *d*.*, *e*.0, *f*.0}, T2 = {*a*.1, *b*.0, c.0, *d*.*, *e*.*, *f*.1}, T3 = {*a*.*, *b*.0, c.0, *d*.0, *e*.*, *f*.*}. And the missing tuples are t1 = {*a*.1, *c*.0, *f*.0}, t2 = {*a*.1, *c*.1, *f*.0}, t3 = {*a*.1, *d*.0, *f*.1}, t4 = {*a*.1, *e*.0, *f*.1}, t5 = {*d*.0, *e*.1, *f*.1}.

As shown in Figure 3-4, we divide all missing tuples into two different groups such that all the tuples in the same group involve the same value of the new parameter. We also divide existing tests into groups in the same way. Note that, while missing tuples must have a value of the new parameter, some existing tests may have “don’t care” value on the new parameter and thus cannot be put into groups, e.g., T3 is not in group while T1 in left group with *f*.0 and T2 in right group with *f*.1. We consider missing tuples as uncolored vertices and existing tests as colored vertices with distinct colors, e.g., T1 has color 1, T2 has color 2 and T3 has color 3.

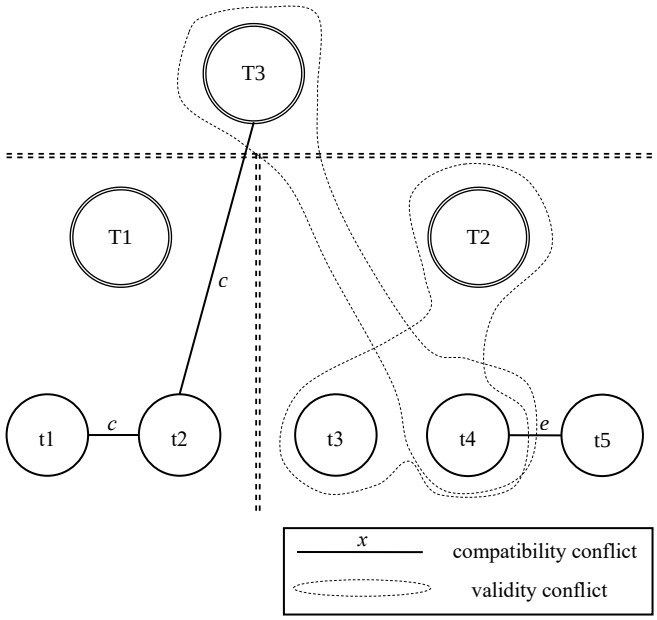

Figure 3-4. Conflict hypergraph reduced by using the inputs of vertical growth

Since we need the Degree of Conflicts (DOC) of each uncolored vertex to get a "good" order for greedy coloring, we have to add compatibility conflicts and validity conflicts into hypergraph.

The compatibility conflicts are represented as edges between two missing tuples, or between one missing tuple and one existing test, if they assign different values on same parameter. (Edge between two existing tests is meaningless since both vertices are already colored.) Note that, any two vertices from different groups always have an edge since they are incompatible on new parameter value, which are hidden in Figure 3-4 for clarity. If there is at least one existing test which can be contested by multiple groups, such as T3, cross-group edges must be counted when computing the DOC. If not, all groups are independent sub-problems.

The above steps are similar to our previous work which uses graph coloring schema without constraints [15]. However, as indicated by motivating examples, we have to add validity conflicts

as hyperedges into hypergraph for making use of constraints. A validity conflict is a combination of vertices which violate constraints. If constraints are given in MFT format, then a hyperedge is a subset of vertices whose values contain an MFT. A hyperedge can be a combination of vertices all of which are missing tuples, or one is an existing test and the others are missing tuples. (Hyperedge having more than one existing test vertices is meaningless since two tests won't combine.)

Note that, there are three fundamental properties of hyperedges in conflict hypergraph:

1. The size of a hyperedge, i.e., the number of vertices it connects, is no less than 2, since a size=1 hyperedge indicates a missing tuple that is invalid by itself and is removed before coloring.

2. A hyperedge should not contain any edge as compatibility conflict, since validity check can only be processed after passing compatibility check, i.e., hyperedge should not be created if it is already incompatible.

3. A hyperedge should not contain any smaller-size hyperedge, i.e., we only use minimum hyperedges to represent validity conflicts from constraints. The reason is that, to check a partial test is valid or not, we only need to check if it contains any minimum hyperedge or not. Otherwise, in the worst case, there will be exponential scale of larger-size combinations of vertices that violate constraints counted as hyperedges, which would incorrectly amplify the impact of constraints on DOC.

According to the above properties of a hyperedge, we can convert MFTs into hyperedges. One MFT may imply multiple hyperedges. For example, in Figure 3-4, MFT1 = {a.1, b.0, d.0, e.0, f.1} can be converted into two hyperedges {T2, t3, t4} and {T3, t4}.

Based on the grouping idea on new parameter value, MFTs can be divided into four categories according to their parameters, which lead to produce corresponding hyperedges in groups of vertices:

TABLE 3-5. TYPES OF MFTS FOR PRODUCING HYPEREDGES

Type of MFTs	Definition	Corresponding Hyperedges	Reason
0	An MFT that contains a param Pm after the new param Pn (i.e., m>n)	None	Any value assignment of P1...Pn (partial test at this step) won't violate a restriction of Pm
1	An MFT that contains only one old param in P1...P(n-1)	None	Any combs of valid t-way target tuples won't violate such a short MFT
2	An MFT that contains at least two old params in P1...P(n-1) and the new param Pn with Pn.value	Produce hyperedges only in the one group of Pn.value	Some tuples in that group of Pn.value may conflict if their combination contains the MFT
3	An MFT that contains at least two old params in P1...P(n-1) and does NOT contain the new param Pn	Produce hyperedges in any group	Some tuples in any Pn group may conflict if their combination contains the MFT

As shown in TABLE 3-5, for vertical growth step of new parameter Pn, we only have to produce hyperedges from type 2 and 3 MFTs in hypergraph, and the new parameter value of MFTs are contained by grouping. Thus, the principle of hyperedge production is that, every vertex in a hyperedge should contribute at least one unique old parameter value to cover the MFT. Otherwise, this hyperedge is definitely not minimum.

The producing of hyperedges as validity conflicts from MFTs is following:

1. For each type 2 (or 3) MFT, in its corresponding group (or for each group), produce minimum hyperedges that contain all old parameter values of the MFT: first, for each old parameter value of the MFT, based on the principle of hyperedge production, collect vertices containing that value as a set, such as V1, V2, V3, ...; then, do cross product on these sets, such as V1×V2×V3×..., save all elements in cross product as hyperedge candidates; finally check candidates to remove those who contain edges or contain smaller-size candidates.

Repeat step 1 until all MFTs are used for producing hyperedges.

For example, assume an MFT is MFT1={*a*.0, *b*.0, *c*.0}; in group of *f*.0, missing tuples are t1={*a*.0, *e*.0, *f*.0}, t2={*b*.0, *e*.0, *f*.0}, t3={*c*.0, *d*.0, *f*.0}, t4={*a*.0, *c*.0, *f*.0}; so vertex sets for old parameter values of MFT1 are V1=[t1, t4], V2=[t2], V3=[t3, t4], cross product V1×V2×V3=[[t1, t2, t3], [t1, t2, t4], [t4, t2, t3], [t4, t2]]. Candidates are c1={t1, t2, t3}, c2={t1, t2, t4}, c3={t4, t2, t3}, c4={t4, t2}, while c2 and c3 should be removed due to they contain c4. So the hyperedges from MFT1 are c1 and c4.

2. Remove hyperedges that contain any smaller-size hyperedge. The reason why we have to do that in global is because, after all MFTs are used, for a single MFT its hyperedges would be minimum, but they may contain a hyperedge from another MFT.

For example, assume MFTs are MFT1={*a*.0, *b*.0, *c*.0}, MFT2={*a*.0, *d*.0}; missing tuples are t1={*a*.0, *e*.0, *f*.0}, t2={*b*.0, *e*.0, *f*.0}, t3={*c*.0, *d*.0, *f*.0}; first MFT1 produces a hyperedge H1={t1, t2, t3}, but later MFT2 produces another hyperedge H2={t1, t3} which is contained by H1. So from the perspective of entire MFTs, H1 is no longer minimum and should be removed.

3.4.2 Compute order for greedy coloring

After generated hypergraph from the inputs of vertical growth, we compute the Degree of Conflicts (DOC) for each vertex, and sort vertices in the non-increasing order of DOC. The DOC of vertex *v* is not the number *n* of edges/hyperedges which touch *v*, but a sum of reciprocal of their sizes S*i* (*i*=1...*n*) as weighting, i.e., DOC(*v*) = $\sum_{i=1}^{n}(2/Si)$. Note that, we let reciprocal times two for making weigh of an edge to be 1.

The sorted vertices can be translated back to a sorted set of missing tuples, in order to replace the set π as an input of vertical growth. It means the vertical growth with optimization should greedily cover missing tuples in DOC order instead of an arbitrary order.

With grouping idea, the formula of DOC(*v*) can be subdivided as DOC(*v*) = the number of edges touch *v* in same group + the number of cross-group edges touch *v* from all vertices in other groups + weighs of hyperedges touch *v*. For example as in Figure 3-4, after the optimization is

proceed, the DOC of missing tuples are: DOC(t1)=1+4+0=5, DOC(t2)=2+4+0=6, $\mathrm{DOC(t3)} = 0 + 3 + \frac{2}{3} = 3\frac{2}{3}$, $\mathrm{DOC(t4)} = 1 + 3 + \frac{2}{3} + \frac{2}{2} = 5\frac{2}{3}$, DOC(t5)=1+3+0=4. So the DOC order is (t2, t4, t1, t5, t3), which makes vertical growth generate 4 tests {T1, t2}, {T2, t4}, {T3, t1}, {t5, t3} to cover all five missing tuples. If using an arbitrary order, e.g., default order (t1, t2, t3, t4, t5), we would get 5 tests {T1, t1}, {T2, t3}, {T3, t5}, {t2}, {t4}.

3.5 Experiment

We implemented in ACTS a new IPOG algorithm that incorporates the Hypergraph Coloring-based (HC) approach to vertical growth with FT-based constraint handling. We report several experiments that were conducted to evaluate the effectiveness of the new IPOG algorithm. In particular, we conducted an experiment that compared ACTS with an existing tool PICT [8]. The experimental results show that the new IPOG algorithm performed better than the original IPOG algorithm in ACTS and PICT for a set of real-life systems.

3.5.1 Experiment design

TABLE 3-6 shows nine real-life systems used in our experiments. These systems have been packaged as example SUTs in ACTS releases for several years.

TABLE 3-6. MODELS AND SOURCES

Software Name	SUT File Name	Software Source
Apache HTTP server	apache.xml	http://httpd.apache.org
Berkeley DB	Berkeley.xml	https://oss.oracle.com/berkeley-db.html

Software Name	SUT File Name	Software Source
Bugzilla bug tracker	bugzilla.xml	https://bugzilla.mozilla.org
GCC compiler	gcc.xml	https://gcc.gnu.org
replace c program	replace.xml	SIR repository, http://sir.unl.edu
SPIN simulator	Spin_S.xml	http://spinroot.com
SPIN verifier	Spin_V.xml	http://spinroot.com
tcas c program	tcas.xml	SIR repository, http://sir.unl.edu
Violet UML editor	Violet.xml	http://sourceforge.net/projects/violet

We adopt the exponential notation to denote parameter model and forbidden tuples, where d^n means that there are n parameters of domain size d, or means that there are n forbidden tuples of size d (size is the number of parameter values in a forbidden tuple). The configurations of these real-life systems are shown in TABLE 3-7. Five systems of the total nine, apache/bugzilla/gcc/Spin_S/Spin_V, are from the benchmarks of Covering Arrays by Simulated Annealing [23]. Their constraints are simply given in Forbidden Tuple format. Another two systems, Berkeley/Violet, are from the benchmarks of Johansen et al. 2011 feature models [20]. Their constraints are also simply given in Forbidden Tuple format. Replace/tcas systems are from our previous research on Input Parameter Modeling [21]. The constraints of replace system are given not in Forbidden Tuple but in Logic Expression format, which requires to be converted into FTs. Note that, MFTs may not be the same as IFTs, when some IFTs can be applied to rule of consensus, i.e., implicit FTs can be derived by these explicit FTs. Only MFTs are appropriate for

using in HC-based optimization, since IFTs may miss some validity conflicts. The method to generate MFTs from IFTs has already been introduced in our other papers [17][18][19].

TABLE 3-7. CONFIGURATIONS OF REAL-LIFE SYSTEMS

Name	Parameter Model	# of Constraints	Input Forbidden Tuples	Minimum Forbidden Tuples
apache	2^{158} 3^{8} 4^{4} 5^{1} 6^{1}	7	$2^{3}3^{1}4^{2}5^{1}$	$2^{3}3^{1}4^{2}5^{1}$
Berkeley	2^{78}	151	$1^{1}2^{144}3^{3}6^{1}7^{2}$	$1^{8}2^{788}3^{21}6^{48}7^{1}$
bugzilla	$2^{49}3^{1}4^{2}$	5	$2^{4}3^{1}$	$2^{4}3^{1}$
gcc	$2^{189}3^{10}$	40	$2^{37}3^{3}$	2^{39}
replace	$2^{4}4^{16}$	36	2^{195}	2^{195}
Spin_S	$2^{13}4^{5}$	13	2^{13}	2^{13}
Spin_V	$2^{42}3^{2}4^{11}$	49	$2^{47}3^{2}$	$2^{56}3^{10}$
tcas	2^{7} 3^{2} 4^{1} 10^{2}	3	2^{3}	2^{6}
Violet	2^{101}	203	$1^{1}2^{191}3^{1}5^{1}6^{1}7^{4}8^{1}9^{2}12^{1}$	$1^{3}2^{783}3^{1}4^{3}5^{3}6^{4}7^{4}9^{1}$

3.5.2 Results and analysis

The experimental environment is set up as the following: OS: Windows 7 64bits, CPU: Intel Dual-Core i5 2.5GHz, Memory: 8 GB DDR3, Java SDK (Since ACTS is a Java tool): Java SE 1.6, Java Max Heap Size: 1024MB.

We generate 2-way and 3-way test sets for the nine subject systems, by using PICT 3.3, ACTS 3.0, and ACTS 3.0-HC. Both PICT and ACTS can accept constraints in logic expression format. Experimental results are shown in TABLE 3-8 and TABLE 3-9. The cells highlighted by gray background indicate the smallest test set sizes produced by PICT and ACTS tools.

TABLE 3-8. RESULT OF 2-WAY TEST GENERATION

Name	**# of Tuples**	**PICT 3.3**		**ACTS 3.0 original IPOG**		**ACTS 3.0-HC new IPOG**	
		size	***Time(s)***	***size***	***Time(s)***	***size***	***Time(s)***
apache	66927	40	0.213	33	0.328	33	0.484
Berkeley	10020	31	0.348	24	15.787	25	17.285
bugzilla	5818	20	0.068	19	0.250	18	0.359
gcc	82770	30	0.178	23	0.468	23	0.780
replace	2261	211	0.116	193	0.921	183	1.295
Spin_S	979	26	0.039	26	0.312	24	0.343
Spin_V	8741	63	0.075	45	0.453	43	0.593
tcas	831	100	0.040	100	0.249	100	0.296
Violet	18820	36	17.099	29	5.912	30	6.770

TABLE 3-9. RESULT OF 3-WAY TEST GENERATION

Name	**# of Tuples**	**PICT 3.3**		**ACTS 3.0 original IPOG**		**ACTS 3.0-HC new IPOG**	
		size	***Time(s)***	***size***	***Time(s)***	***size***	***Time(s)***
apache	8085958	202	136.163	173	12.184	171	12.964
Berkeley	423992	113	4.216	94	16.957	93	20.264
bugzilla	202683	70	0.660	68	0.343	68	0.499
gcc	11131894	134	138.207	108	14.414	106	19.172
replace	41678	923	1.159	975	1.981	864	3208.434
Spin_S	12835	113	0.122	98	0.296	94	0.468
Spin_V	369976	345	4.649	286	0.827	270	26.380
tcas	8929	409	0.129	405	0.250	405	0.327
Violet	1148263	167	29.955	149	8.253	N/A	OOM

OOM = Out of Memory Java exception

TABLE 3-8 shows that, for 2-way test generation, compared with the original IPOG algorithm, the new IPOG algorithm significantly reduces test set size for one of the nine real-life systems, and slightly reduces test set size for three of the nine systems, while slightly increases test set size for two of the nine systems. Both algorithms produce the same test set size for the

other three systems. Even the vertical growth phase in the original IPOG algorithm is nearly to be optimal for 2-way testing ignoring constraints [15], if there are strict constraints that introduce many conflicts, e.g., in replace system, the new IPOG algorithm is more greedy for using 10 less test cases to cover tuples.

TABLE 3-9 shows that, for 3-way test generation, compared with the original IPOG algorithm, the new IPOG algorithm reduces test set size for six of the nine real-life systems, especially for replace/Spin_V systems which are cut off 111 or 16 tests compared to original ACTS test set sizes; but it fails on the last system due to Out of Memory exception. Both algorithms produce the same test set size for the other two systems. These results demonstrate that our new vertical growth algorithm is effective for *t*-way test generation when *t* is greater than two. Note that, for replace system, the new IPOG algorithm consumes much more time than original IPOG due to the overhead of hyperedges production.

3.6 Related Work

In our past work, we presented a Graph Coloring-based improvement [15] on IPOG's vertical growth phase. Our GC-based approach cannot handle constraints. In contrast, our approach presented in this paper introduces the notion of hypergraph that allows validity conflicts to be extracted from constraints.

Hallé et al. [16] presented two reductions of t-way test generation problem to graph coloring and hypergraph vertex covering, respectively. The graph coloring reduction follows the same idea

of our GC-based approach in [15]. It links the minimal number of tests to the chromatic number of some graph, where vertices are t-way target tuples and edges are the pairs of vertices that cannot be true at the same time. Like GC-based approach, it doesn't support constraints that are violated only by a combination of more than two target tuples.

The hypergraph vertex covering reduction in [16] is different from our HC-based approach. It considers each vertex as a possible test. Thus the set of its vertices is the exhaustive set of all possible tests, i.e., all possible combinations of parameter values. It considers each hyperedge to be a subset of tests that covers the same group of target tuples. To cover all the target tuples is to find a vertex cover that covers each hyperedge. A hyperedge is covered if one of its vertices is covered. Compared to hypergraph coloring, which use a number of colors (chromatic number as lower bound) to cover all vertices, hypergraph vertex covering use a subset of vertices (minimum hitting set as lower bound) to cover all the hyperedges. Note that given a system with *n* parameters all with domain size *k,* hypergraph vertex covering uses exhaustive valid tests as vertices whose number is $O(k^n)$. The resulting hypergraph can be much larger than the hypergraph for hypergraph coloring where vertices represent t-way target tuples whose number is $O\left(\binom{n}{t} \times k^t\right)$.

For handling constraints during combinatorial test generation, there are two major approaches, constraint solving-based approaches, and forbidden tuple-based approaches. SAT solvers are used in many constraint solving-based approaches. Garvin et al. integrated a SAT solver into a

meta-heuristic search algorithm for constrained combinatorial test generation [22]. Cohen et al. integrated a SAT solver into an AETG-like test generation algorithm [24]. In our early work [5], we integrated a CSP solver into the IPOG algorithm and proposed several optimizations to improve its performance.

Forbidden tuple-based approaches handle constraints based on the notion of forbidden tuple. Some forbidden tuple-based approaches such as *T*tuples [25] require all forbidden tuples be explicitly listed. In *T*tuples, a partial test is treated as valid if it does not violate constraints involving only fixed parameters. This approach can generate invalid tests if some implicit forbidden tuples can be derived from explicitly given forbidden tuples. PICT [9] translates constraints to a set of exclusions (i.e., forbidden tuples), derives implicit ones from existing exclusions, and then uses the complete set to ensure validity of tests. However, its technical details of how to generate complete set of exclusions are not discussed. In our early work [17][18][19], we presented the MFT generation algorithm to derive and simplify implicit forbidden tuples.

3.7 Conclusion and Future Work

In this paper, we present an optimization on the vertical growth phase of the IPOG algorithm. In this optimization we create a hypergraph model that captures the conflict relationship among missing tuples and existing tests after horizontal growth. We focus on the impact of constraints on vertical growth. Since constraints may introduce conflicts that involve more than two tuples, we

show how to represent such higher-degree conflicts as hyperedges and how to use hyperedges for optimizing vertical growth. Doing so allows the vertical growth problem to be reduced to a classical NP-hard problem "Minimum Vertex Coloring" on a hypergraph. Given some missing tuples, an order of missing tuples is determined based on the degree of conflicts for each missing tuple. The missing tuples are then greedily covered in this order. Specifically, the higher the degree of conflicts of a missing tuple, the earlier it is covered. The experimental results show that the optimization can further reduce the number of tests that are generated by the IPOG algorithm for a number of real-life systems.

In the future, we will consider the overhead of producing a hypergraph, mainly on converting MFTs into hyperedges. Since the number of validity conflicts is growing exponentially with strength t and the size of MFT, we will explore the use of heuristics to extract validity conflicts from constraints, in order to cut down the optimization time and space cost for high-strength t-way testing, and for systems with large-size MFTs.

3.8 Acknowledgment

This work is partly supported by a research grant (70NANB15H199) from Information Technology Laboratory of National Institute of Standards and Technology (NIST).

DISCLAIMER: NIST does not endorse or recommend any commercial product referenced in this paper or imply that a referenced product is necessarily the best available for the purpose.

3.9 References

[1] Y. Lei, and K.C. Tai. “In-parameter-order: A test generation strategy for pairwise testing.” In High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International, pp. 254-261. IEEE, 1998.

[2] K.C. Tai, and Y. Lei. “A test generation strategy for pairwise testing.” IEEE Transactions on Software Engineering 28, no. 1 (2002): 109-111.

[3] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, and J. Lawrence. “IPOG: A general strategy for t-way software testing.” In Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the, pp. 549-556. IEEE, 2007.

[4] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, and J. Lawrence. “IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing.” Software Testing, Verification and Reliability 18, no. 3 (2008): 125-148.

[5] L. Yu, Y. Lei, M. Nourozborazjany, R.N. Kacker, and D. R. Kuhn. “An efficient algorithm for constraint handling in combinatorial test generation.” In Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, pp. 242-251. IEEE, 2013.

[6] ACTS (Advanced Combinatorial Testing System), http://csrc.nist.gov/acts/

[7] L. Yu, Y. Lei, R.N. Kacker, and D.R. Kuhn. “Acts: A combinatorial test generation tool.” In Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, pp. 370-375. IEEE, 2013.

[8] PICT (Pairwise Independent Combinatorial Testing tool), http://blogs.msdn.com/b/nagasatish/archive/2006/11/30/pairwise-testing-pict-tool.aspx

[9] J. Czerwonka. “Pairwise testing in the real world: Practical extensions to test-case scenarios.” In Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer, pp. 419-430. 2006.

[10] D.W. Matula, G. Marble, and J.D. Isaacson. “Graph coloring algorithms.” Graph theory and computing (1972): 109-122.

[11] S. Skiena. “Finding a Vertex Coloring.” §5.5.3 in Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Reading, MA: Addison-Wesley, pp. 214-215, 1990.

[12] B. Manvel. “Extremely greedy coloring algorithms.” Graphs and Applications (1985): 257-270.

[13] L. Kučera. “The greedy coloring is a bad probabilistic algorithm.”d Journal of Algorithms 12, no. 4 (1991): 674-684.

[14] C. Berge. Hypergraphs: combinatorics of finite sets. Vol. 45. Elsevier, 1984

[15] F. Duan, Y. Lei, L. Yu, R.N. Kacker, and D.R. Kuhn. “Improving IPOG's vertical growth based on a graph coloring scheme.” In Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on, pp. 1-8. IEEE, 2015.

[16] S. Hallé, E. La Chance, and S. Gaboury. “Graph Methods for Generating Test Cases with Universal and Existential Constraints.” In IFIP International Conference on Testing Software and Systems, pp. 55-70. Springer International Publishing, 2015.

[17] L. Yu, F. Duan, Y. Lei, R.N. Kacker, and D.R. Kuhn. “Combinatorial test generation for software product lines using minimum invalid tuples.” In 2014 IEEE 15th International Symposium on High-Assurance Systems Engineering, pp. 65-72. IEEE, 2014.

[18] L. Yu, F. Duan, Y. Lei, R.N. Kacker, and D.R. Kuhn.”Constraint handling in combinatorial test generation using forbidden tuples.” In Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on, pp. 1-9. IEEE, 2015.

[19] D.R. Kuhn, R. Bryce, F. Duan, L.S. Ghandehari, Y. Lei, and R.N. Kacker. “Combinatorial Testing: Theory and Practice.” Advances in Computers 99 (2015): 1-66.

[20] M.F. Johansen, Ø. Haugen, and F. Fleurey. “Properties of realistic feature models make combinatorial testing of product lines feasible.” In International Conference on Model Driven Engineering Languages and Systems, pp. 638-652. Springer Berlin Heidelberg, 2011.

[21] L.S. Ghandehari, M.N. Bourazjany, Y. Lei, R.N. Kacker and D.R. Kuhn, “Applying Combinatorial testing to the Siemens Suite”, In Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, pp. 362-371, 2013.

[22] B.J. Garvin, M.B. Cohen, and M.B. Dwyer. “An improved meta-heuristic search for constrained interaction testing.” In Search Based Software Engineering, 2009 1st International Symposium on, pp. 13-22. IEEE, 2009.

[23] B.J. Garvin, M.B. Cohen, and M.B. Dwyer. “Evaluating improvements to a meta-heuristic search for constrained interaction testing.” Empirical Software Engineering 16, no. 1 (2011): 61-102.

[24] M.B. Cohen, M.B. 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, pp. 633–650, 2008.

[25] A. Calvagna, and A. Gargantini. “T-wise combinatorial interaction test suites construction based on coverage inheritance.” Software Testing, Verification and Reliability 22, no. 7 (2012): 507-526.

AN APPROACH TO T-WAY TEST SEQUENCE GENERATION WITH CONSTRAINTS

Feng Duan, Yu Lei, Raghu N. Kacker, and D. Richard Kuhn. “An Approach to T-Way Test Sequence Generation With Constraints.” In 12th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 241-250. IEEE, 2019.

CHAPTER 4.

An Approach to T-way Test Sequence Generation With Constraints

Abstract—In this paper we address the problem of constraint handling in t-way test sequence generation. We develop a notation for specifying sequencing constraints and present a t-way test sequence generation that handles the constraints specified in this notation. We report a case study in which we applied our notation and test generation algorithm to a real-life communication protocol. Our experience indicates that our notation is intuitive to use and allows us to express important sequencing constraints for the protocol. However, the test generation algorithm takes a significant amount of time. This work is part of our larger effort to make t-way sequence testing practically useful.

Keywords—*Test sequence generation; Sequencing constraint; T-way sequence coverage; Sequence testing; Event-based testing; Combinatorial testing*

4.1 Introduction

Many systems, e.g., interactive systems [1], event-driven systems [2] and communication protocols [3], exhibit sequencing behavior, where a sequence of events is exercised during each execution and the order in which the events occur could significantly affect the system behavior. To test these systems, we need to generate test sequences, in addition to test data.

T-way sequence testing applies the notion of *t*-way coverage to test sequence generation [4]. Informally, given any *t* events, if they could be exercised in a given order, there must exist at least one test sequence in which these events are exercised in this order. Doing so allows us to test all possible interactions between any *t* events. Thus, *t*-way sequence testing can expose faults that are caused by interactions between no more than *t* events.

One important problem in *t*-way sequence testing is dealing with sequencing constraints [5], i.e., restrictions on the order of events that must be satisfied for a test sequence to be valid. The technical challenge is two-fold. First, a notation is needed to specify sequencing constraints. This notation must be easy to use and have the power to express commonly encountered constraints. Second, a test generation algorithm must be developed to handle sequencing constraints. Compared to constraints on data values, sequencing constraints can be more difficult to handle. This is because the space that needs to be searched in the evaluation process for sequencing constraints can be much larger due to the extra dimension, i.e., order of events.

Recent years have seen significant progress on *t*-way test data generation, but not on *t*-way test sequence generation [6]. Kuhn et al. [4] presented an approach to generate SCAs (Sequence Covering Arrays) for testing special systems. Their approach requires each event occurs exactly once in each test sequence, and only supports one type of constraint. Yu et al. [7][8] presented another approach to *t*-way test sequence generation based on a Labeled Transition System (LTS) model that captures system behavior. An LTS model is similar to a finite state machine where the

sequencing constraints are implicitly encoded in the state transitions. However, as an operational model, LTS is at a very low level of abstraction and requires a lot of details on how a system operates in terms of states and transitions. As a result, LTS models are seldom available in practice.

In this paper we develop a notation for specifying sequencing constraints and present a *t*-way test sequence generation algorithm that handles constraints specified in this notation. Our notation adopts an event-oriented framework. It defines a small set of basic operators that capture several fundamental orderings that could happen between two events. These operators can be nested, if necessary, to specify the sequencing behavior between multiple events. Our notation is at a higher level of abstraction than an operational model such as LTS. Also we believe that since we deal with sequences of events, an event-oriented notation is more intuitive than a state machine-based notation.

Our test sequence generation algorithm employs a greedy strategy in which each test sequence is generated such that a maximal number of *t*-way target sequences can be covered in the sequence. A *t*-way target sequence is a sequence of *t* events that could be covered in the given order, i.e., the order in which they appear in the sequence. Each test sequence is generated in two phases, including the starting phase and the extension phase. In the starting phase, we generate a starting sequence that is guaranteed to cover at least one target sequence. In the extension phase, we keep extending the test sequence until no extension is possible. At each extension, we append

to the test sequence an event that covers the most *t*-way target sequences that are yet to be covered.

We report a case study in which we applied our approach to a communication protocol, i.e., IEEE 11073-20601 [10]. This protocol is used to exchange data between Personal Health Devices (PHDs), e.g., smart scales, and computing devices, e.g., desktop computers. We identified a set of 9 sequencing constraints and wrote them using our notation. We generated a set of 24 test sequences that achieve 2-way sequence coverage while satisfying all the sequencing constraints. Our experience indicates that our notation is intuitive to use and allows us to express important sequencing constraints for this protocol. However, while our algorithm allows us to generate *t*-way test sequence set, it is computationally expensive. For example, it takes 5 and half hours for generating test sequences with lengths up to 9, and 2 days with lengths up to 10. While this is partly due to the nature of the problem, we believe there are opportunities for optimization which we will explore in our future work.

We focus on positive testing in this paper. That is, we generate test sequences that satisfy all the sequencing constraints. However, sequencing constraints are also useful for negative testing. For example, test sequences could be generated for negative testing that violate constraints in a systematic manner.

The remainder of the paper is organized as follows. Section 4.2 explains our basic idea with a motivating example. Section 4.3 presents the syntax and semantics of our notation to specify

sequencing constraints. We also introduce two other types of constraints, i.e., repetition and length constraints, which are used to control test sequence length. Section 4.4 first presents the definitions of *t*-way target and test sequences, and then describes our test generation algorithm that handles constraints. Section 4.5 presents the results of our case study on the PHD protocol, including the sequencing constraints identified and some statistics on the test sequences generated. Section 4.6 discusses related work on *t*-way test sequence generation. Section 4.7 provides concluding remarks and our plan for future work.

4.2 Motivating Example

In this section, we use File API as a motivating example to show the basic idea of our work, including how to model the sequencing behavior of a system in terms of events and constraints, and how to generate a set of test sequences for t-way sequence coverage.

4.2.1 Model Sequencing Behavior

A sequencing model M = <E, C> consists of two components: (1) E: a set of events that could be exercised in a system execution; (2) C: a set of constraints that restrict the occurrences of these events in a system execution.

In File API, there are four major file operations, including *open*, *close*, *read*, and *write*. In the sequencing model, each of these operations is modeled as an event. Thus, E = {*open*, *close*, *read*, *write*}.

Based on the semantics of file operations, the following constraints can be identified in terms of the order in which these events could be exercised. These constraints are referred to as sequencing constraints.

(1) The first event of a test sequence must be *open*.

(2) The file must be open before *read*, *write*, or *close*.

(3) The last event of a test sequence must be *close*.

We introduce an event-oriented notation to model the above constraints. To model the first constraint, we specify that the *open* event must happen before all the other three events. More precisely, in a test sequence, whenever there is a *read*, *write*, or *close* event *e*, there must exist an *open* event that is exercised before *e*. To model the second constraint, we specify that *open* must happen before *read*, *write* or *close* and there shall be no *close* in between. To model the third constraint, we specify that the *close* event must happen after all other three events. More precisely, in a test sequence, whenever there is an *open*, *read*, or *write* event *e*, there must exist a *close* event that is exercised after *e*.

In addition to sequencing constraints, we introduce two other types of constraint, namely repetition and length constraints, to control the length of a test sequence. A repetition constraint specifies the number of times a certain event could be repeated in a test sequence. For example, we could specify that *open*/*close* could only occur once in a test sequence. A length constraint specifies the minimum and/or maximum length of a test sequence.

In Section 4.3, we introduce a formal notation to specify the three types of constraints in detail.

4.2.2 Test Sequence Generation

After a sequencing model is specified, a test sequence set can be generated to achieve *t*-way sequence coverage. Recall that *t*-way sequence coverage requires that every *t*-way (target) sequence, i.e., every sequence of *t* events that could be exercised in the given order, consecutively or not, be exercised so by at least one test sequence.

For example, for File API, the set of 2-way sequences is {<*open*, *open*>, <*open*, *close*>, ..., <*read*, *write*>, ..., <*write*, *write*>} (the set size = 4^2 = 16). Note that the existence of constraints may make some of these sequences *uncoverable*, i.e., they cannot be covered by any test sequence that satisfies all the constraints. In this paper, we use <e_1, e_2, ...> to represent target sequences, and use [e_1, e_2, ...] to represent test sequences. A 2-way target sequence <e_1, e_2> is covered by a test sequence in the form of [..., e_1, ..., e_2, ...].

If we specify some repetition constraints such that no event could be repeated in a test sequence, in addition to the three sequencing constraints mentioned earlier, there is a total of seven 2-way target sequences {<*open*, close>, <*open*, *read*>, <*open*, *write*>, <*read*, *close*>, <*read*, *write*>, <*write*, *close*>, <*write*, *read*>}. A greedy algorithm can be used to generate a 2-way test sequence set such that all the test sequences satisfy all the constraints, and every target sequence is covered by at least one test sequence.

The details of the greedy algorithm are presented in Section 4.4. The following example illustrates the basic idea of the algorithm:

1. We first construct a starting sequence that covers at least one target sequence. We begin with an empty test sequence []. The only possible event that could be added is *open* due to sequencing constraint (1). The resulting sequence is [*open*].

2. We further extend [*open*]. There are three possible choices: [*open*, *read*]/[*open*, *write*]/[*open*, *close*]. Note that [*open*, *open*] is not allowed due to the repetition constraint. All of the three choices cover one target sequence. We choose [*open*, *read*] as our starting sequence.

3. Now we try to extend the starting sequence. We append to the sequence one event at a time. We first append *write*, followed by *close*, as these two events allow the most target sequences to be covered. The resulting sequence is [*open*, *read*, *write*, *close*], which cannot be further extended.

4. We repeat the above process to generate additional test sequences until all the remaining target sequences are covered.

The final 2-way test sequence set consists of two test sequences {[*open*, *read*, *write*, *close*], [*open*, *write*, *read*, *close*]}. These two sequences satisfy all the constraints and cover all of the seven 2-way target sequences.

4.3 Notation for Constraint Specification

Our notation supports three types of constraints, including repetition, length and sequencing constraints. Recall that repetition and length constraints are used to control the length of a test sequence.

First, we introduce the syntax and semantics of repetition and length constraints:

- A repetition constraint is in the form of "$e.\# \leq r$", denoting that, in a test sequence, an event *e* could occur no more than *r* times. A default repetition constraint can be specified in the form of "$\# \leq r$", denoting that no event could occur for more than *r* times. The default constraint can be overridden by an event-specific constraint.
- A length constraint is in the form of "TOTAL_LEN ≥ *min*" or "TOTAL_LEN ≤ *max*", denoting that, the total length of a test sequence should be greater than or equal to *min*, or/and smaller than or equal to *max*.

By default, "$\# \leq 1$" is given to ensure the termination of test sequence generation. That is, by default, each event is only allowed to appear once in a test sequence.

The rest of this section is focused on the syntax and semantics of sequencing constraints.

4.3.1 Syntax of Sequencing Constraints

The syntax of sequencing constraints is specified in BNF (Backus–Naur form) as shown in Fig. 4-1.

<sequencing constraint> ::= <sequencing expression>

| <sequencing constraint> || <sequencing constraint>

| <sequencing constraint> && <sequencing constraint>

| (<sequencing constraint>)

<sequencing expression> ::=

<sequencing expression> <general sequencing operator> <events>

| <events> <general sequencing operator> <events>

| <events> <immediate sequencing operator> <events>

<events> ::= <event> | <event set>

| <always sequencing operator> <event>

| <always sequencing operator> <event set>

<always sequencing operator> ::= "_"

<immediate sequencing operator> ::= "*–" | "–*" | "~"

<general sequencing operator> ::= "*..." | "...*" | "·~·"

Figure 4-1. BNF of sequencing constraints

There are two types of operators: Boolean and sequencing operators. Sequencing operators can be divided into three groups: immediate operators, general operators, and always operator.

Note that <event set> is a set of events. That is, we allow event sets, as well as individual events, in a constraint expression. In this paper, we will use the notation of $\{e_1, e_2, \ldots, e_n\}$ to denote an event set, where $e_1, e_2, \ldots, e_n$ are individual events. The reason why this is allowed is explained in Section 4.3.2.

A sequencing constraint can be derived from this syntax as shown in Fig. 4-2.

<sequencing constraint>

=> <sequencing expression>

=> <sequencing expression> <general sequencing operator> <events>

=> <events> <general sequencing operator> <events> <general sequencing operator> <events>

=> <always sequencing operator> <event> <general sequencing operator> <event> <general sequencing operator> <event set>

=> *open* ·∼· *close* ···* {*read, write, close*}

Figure 4-2. Derivation of a sequencing constraint from BNF

Note that the precedence of the operators is defined from highest to lowest as follows: unary sequencing operator, binary sequencing operators, (), && and ||.

4.3.2 Semantics of Sequencing Operators

TABLE 4-1. INFORMAL EXPLANATION OF SEQUENCING OPERATORS

Sequencing operator	Explanation
_e_1 (or $\underline{e_1}$)	e_1 **always** happens
e_1 *– e_2	If e_1 happens, then e_2 **must immediately happen after** e_1
e_1 –* e_2	If e_2 happens, then e_1 **must immediately happen before** e_2
$e_1 \sim e_2$	e_2 **never immediately happens after** e_1 (or e_1 **never immediately happens before** e_2)
e_1 *··· e_2	If e_1 happens, then e_2 **must happen after** e_1, **but not necessarily immediately happen after** e_1
e_1 ···* e_2	If e_2 happens, then e_1 **must happen before** e_2, **but not necessarily immediately happen before** e_2
e_1 ·∼· e_2	e_1 **never happens before** e_2 (or e_2 **never happens after** e_1)

TABLE 4-1 lists all the sequencing operators in our notation and provides an informal explanation of each operator. We make the following notes about the symbols used to represent the operator:

(a) _: This indicates that the event always happens. (In this paper, this operator is shown as an underline, e.g., $\underline{e_1}$.)

(b) −/···: Both indicate the left event happens before the right event. However, − requires that the two events are next to each other, whereas ··· does not.

(c) ~/·~·: Both indicate the left event never happens before the right event. However, ~ only requires that the two events do not happen next to each other, whereas ·~· requires that the right event cannot happen one after the left event.

(d) *: It indicates that, when it appears left (or right) to the operator, the constraint applies only if the left (or right) event happens.

Recall that we allow a sequencing operator to be applied to a set of events. For example, "E1 *− E2" denotes that, immediately after any event in set E1, an event in set E2 must happen. The reason why this is necessary is that Boolean operators on sequencing constraints with individual events cannot specify some constraints. For example, "e_1 *− {e_2, e_3}" ≠ "e_1 *− e_2 || e_1 *− e_3". Given a test sequence [...e_1, e_2, ..., e_1, e_3, ...] containing two occurrences of event e_1, "e_1 *− e_2" is not satisfied on the second occurrence of e_1, while "e_1 *− e_3" is not satisfied on the first occurrence of e_1. Thus, none of the two constraints are satisfied. However, the constraint "an event belongs to {e_2, e_3} must happen after e_1" denoted by "e_1 *− {e_2, e_3}" is satisfied on both occurrences of e_1.

In the following, we use an automaton to formally define the semantics of each sequencing operator. Note that in each automaton, ¬ e indicates any event other event e; ∀e indicates any event; $e_1 \vee e_2$ indicates e_1 or e_2. Here we assume that e_1 and e_2 are single events and different (i.e., $e_1 \neq e_2$). The automata for notation in which e_1, e_2 are event sets and $e_1 \wedge e_2$ would be not empty (i.e., $e_1 \wedge e_2 \neq \emptyset$), are given in the next Chapter 5.

4.3.2.1 Always sequencing operator

(1) _ e_1 (a.k.a. $\underline{e_1}$)

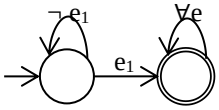

Figure 4-3. Semantics of "_ e_1"

Fig. 4-3 shows that, given an input sequence, if an occurrence of event e exists, the sequence should be accepted. Otherwise, it is rejected.

4.3.2.2 Immediate sequencing operators

(2) e_1 *− e_2

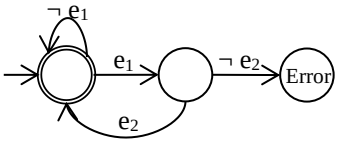

Figure 4-4. Semantics of "e_1 *− e_2"

Fig. 4-4 shows that given an input sequence, if every occurrence of event e_1 is immediately followed by an occurrence of event e_2, the sequence should be accepted. Otherwise, it is rejected.

(3) $e_1 \text{ } -\!\!* \text{ } e_2$

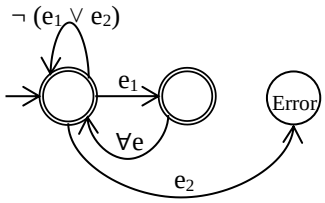

Figure 4-5. Semantics of "$e_1 \text{ } -\!\!* \text{ } e_2$"

Fig. 4-5 shows that given an input sequence, if any occurrence of event e_2 exists that is NOT immediately after an occurrence of event e_1, the sequence should be rejected. Otherwise, it is accepted.

(4) $e_1 \sim e_2$

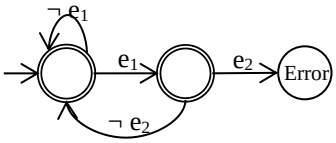

Figure 4-6. Semantics of "$e_1 \sim e_2$"

Fig. 4-6 shows that, given an input sequence, if any occurrence of event e_2 exists immediately after an occurrence of event e_1, the sequence should be rejected. Otherwise, it is accepted.

4.3.2.3 General sequencing operators

The automaton of each general operator is similar to the automaton of the corresponding immediate operator, but in general simpler. This is because the semantics of the immediate operators are stricter, except that ·∼· is stricter than ∼.

(5) e_1 *··· e_2

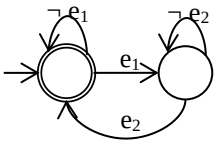

Figure 4-7. Semantics of "e_1 *··· e_2"

(6) e_1 ···* e_2

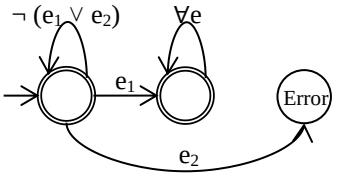

Figure 4-8. Semantics of "e_1 ···* e_2"

(7) e_1 ·∼· e_2

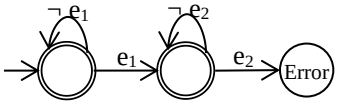

Figure 4-9. Semantics of "e_1 ·∼· e_2"

General sequencing operators can be nested in our notation. The semantics of a nested expression "<sequencing expression> <general sequencing operator> <events>", where <sequencing expression> is the nesting expression, can be defined in a recursive manner. As an example, consider "B ···* e_3", where B = "$\underline{e_1}$ ·∼· e_2". This nested expression can be written as "$\underline{e_1}$ ·∼· e_2 ···* e_3". It denotes that if event e_3 happens, B must be satisfied by a subsequence before e_3 in which "e_1 always happens, and e_2 never happens after e_1". In other words, before each occurrence

of e_3, there must be an occurrence of e_1 to satisfy B, so that e_2 never happens between this occurrence of e_1 and the occurrence of e_3.

Recall that a sequencing operator may involve an event set E. The semantics of each operator with a set of events can be specified using the corresponding automaton with individual events, except that we need to replace "event e" with "any event in E".

4.3.3 Example

We can use our constraint notation to represent the sequencing constraints identified in Section 4.2 as follows:

(1) "*open* ···* {*close, read, write*}": This constraint says that, if there exists an event *close, read* or *write*, there must exist an *open* event before this event. This ensures that *open* is the first event.

(2) "*open* ·~· *close* ···* {*read, write, close*}": This constraint says that if there exists an occurrence of event *read*, *write*, or *close*, there **must** exist an occurrence of event *open* **before** it, and the subsequence separated by the two occurrences satisfies "after this occurrence of *open*, before this occurrence of *read*, *write*, or *close*, an occurrence of *close* **never** exist". This ensures that the file is open before it is read, written, or closed.

(3) "{*open, read, write*} *··· *close*": This constraint says that, after there exists an *open*, *read* or *write* event, a *close* event must exist. This ensures that *close* is the last event.

4.4 Approach

4.4.1 Basic Concepts

Informally, a target sequence is a sequence of events that needs to be covered. A test sequence is a sequence of events that can be executed by the subject system. Target sequences are covered by test sequences to satisfy *t*-way sequence coverage. In other words, target sequences are test requirements, i.e., entities that must be covered to achieve *t*-way sequence coverage, while test sequences are test cases that cover test requirements. T-way sequence coverage requires that for every target sequence of *t* events, if they could be exercised in the given order, they must be exercised so in at least one test sequence.

In the following, we formalize these concepts from the perspective of test sequence generation, without considering the semantics of the subject system. Let M = <E, C> be the sequencing model of the subject system.

Definition 1. A sequence Q of events is *valid* if it satisfies all the constraints in C; otherwise it is invalid.

In this paper we focus on positive testing. Thus, every test sequence must be a valid sequence. Also, every valid sequence can be used as a test sequence.

Definition 2. A sequence Q of events is *extendable* if it is a proper prefix of another sequence of events that is valid.

Note that an extendable sequence itself may or may not be valid.

Definition 3. A sequence Q of events *covers* another sequence Q' of events if all the events in Q' appear in Q in the same order as they appear in Q'.

In the above definition, it is important to note that the events in Q' do not have to appear consecutively in Q. For example, a partial test sequence [*open*, *read*, *write*] covers three 2-way sequences: <*open*, *read*>, <*open*, *write*> and <*read*, *write*>.

Definition 4. A *t*-way target sequence Q is a *t*-way sequence that can be covered by at least one test sequence.

Note that not every *t*-way sequence is a target sequence. Consider the File API example. If a repetition constraint requires that no event can be repeated, then 2-way sequences <*open*, *open*> and <*close*, *open*> cannot be covered by any test sequence. Thus, these sequences are not 2-way target sequences.

Definition 5. Let Π be the set of all the *t*-way target sequences. A *t*-way test sequence set Σ is a set of test sequences such that for $\forall \pi \in \Pi$, $\exists Q \in \Sigma$ such that Q covers π.

Considering the motivating example, a set of two test sequences {[*open*, *read*, *write*, *close*], [*open*, *write*, *read*, *close*]} covers all 2-way target sequences <*open*, close>, <*open*, *read*>, <*open*, *write*>, <*read*, *close*>, <*read*, *write*>, <*write*, *close*> and <*write*, *read*>.

Note that the above definitions are similar to our earlier work in [7], which is however based on LTS.

4.4.2 Main Idea

```
Input: (a) A sequencing model M = (E, C), where E is a set of events and C is a set of constraints,
and (b) a test strength t
Output: A t-way test sequence set Σ
{
   // Step 1: target sequence (candidate) generation
1.  let Π be {π = <e1, e2, …, et> | ei∈E}
   // Step 2: test sequence generation
2.  let Σ be an empty set
3.  while (Π is not empty) {
      // Step 2.1: starting phase
4.    create a starting test sequence Q such that (a) Q covers at least one target sequence in Π; and
          (b) Q is valid or extendable
5.    if (Q cannot be created)
6.      break
7.    remove from Π the target sequences covered by Q
      // Step 2.2: extension phase
8.    while (Q is extendable) {
9.      append an event e in E to Q such that (a) Q.e covers the most target sequences in Π; and
            (b) Q.e is valid or extendable
10.     Q = Q.e
11.     remove from Π the target sequences covered by Q
12.   }
13.   add Q into Σ
14. }
15. return Σ
}
```

Figure 4-10. Algorithm GenTestSeqs

Fig. 4-10 shows our test generation algorithm. The algorithm consists of three major steps. The first step is to generate target sequence candidates. Note that not every sequence in Π is a

target sequence, as some sequences in Π may not be covered by any test sequence. As discussed later, the second step guarantees that all the target sequences in Π will be covered. Thus, after the second step, the remaining sequences in Π cannot be covered by any test sequence and are not target sequences. We could check whether every sequence in Π is a target sequence and remove those that are not prior to the second step. This, however, is expensive and redundant.

In the second step, we generate test sequences to cover all the target sequences. We first create a starting test sequence Q to cover at least one remaining target sequence. A starting test sequence must be valid or extendable. This is necessary to ensure termination. If such a test sequence cannot be created, all the target sequences in Π have already been covered, and the algorithm terminates. Otherwise, we extend Q by appending events one by one. Each time we select an event that covers the most target sequences in Π. When no event can be appended to Q, Q becomes a *complete* (valid and not-extendable) test sequence, and we add Q into the resulting test set and create another starting test sequence. We continue to do so until we cannot find a starting sequence that covers at least one sequence in Π.

We call the phase of creating a starting test sequence as a starting phase in Lines 4 - 7, and the phase of extending a test sequence to be complete as an extension phase in Lines 8 - 13.

4.4.3 Validity and Extensibility Check

In this part we discuss how to check if a test sequence is valid and if a test sequence is extendable. These are two important checks performed in our algorithm.

1. Validity check:

Recall that there are three types of constraints, sequencing, repetition, and length constraints. Given a test sequence Q, we first check whether it satisfies all the repetition and length constraints, which is accomplished by counting its number of events. Next we check whether it satisfies all the sequencing Constraints, which is more complicated and is described as follows.

As indicated by the BNF grammar of Section 4.3, there are two types of constraints for solving: basic and nested.

(1) A basic expression "*e* <sequencing operator> *e*" only involves two events (or event sets) in sequence, such as "e_1 *− e_2". Based on basic temporal logic (as corresponding automaton), the basic expression is true on Q if and only if Q is accepted by our automaton. Note that, our Sequencing Constraint Solver is only applicable to test sequence, i.e., not target sequences whose validity need to be checked differently.

(2) A nested expression "B <sequencing operator> *e*" involves more than two events in sequence, since B is another sequencing expression, such as "$\underline{e_1}$ ·∼· e_2 *··· e_3", (i.e., "B *··· e_3", B = "$\underline{e_1}$ ·∼· e_2"). Automatons will be recursively called by the nested structure. The Boolean result of the nested expression "B *··· e_3" on Q is decided by "if B is true on a subsequence of Q, whether e_3 happens after the subsequence". Thus, the global Boolean result is that "if 'e_1 always happens and e_2 never happens after e_1' is true, whether e_3 happens after e_1".

2. Extensibility check:

Fig. 4-11 shows our algorithm for checking whether a sequence is extendable. The algorithm employs a recursive DFS (Depth-First Search) strategy. Note that in order to prevent infinite extension, we set a default repetition constraint which requires every event be repeated no more than *t* times, where *t* is the coverage strength, if the user does not specify any length constraint to restrict the maximum length of a sequence. Thus, a maximum length could always be derived from the repetition and length constraint.

```
Boolean isExtendable(Q, E, C)
{
   let max_length be the maximum length implied by repetition/length constraints
   if (Q.length >= max_length)
      return false
   for (each event e in E) {
      set Q' to be Q.e
      if (isValid(Q', C))
         return true
      else if (isExtendable(Q', E, C))
         return true
   }
   return false
}
```

Figure 4-11. Extensibility check algorithm

4.4.4 Test Sequence Generation

In this part, we discuss two main challenges of our generation approach shown in Fig. 4-10.

The first challenge of our generation approach is that, due to the limitation of our automatons which are only available for consecutive sequence, we cannot directly check the validity of target sequences.

As indicated in Line 1, we enumerate all possible permutations (with repetition) of any *t* events as *t*-way target sequence candidates. The same event could be exercised for up to *t* times in a permutation. Recall that some of these candidates cannot be covered by any test sequence, while others can be covered.

Our solution is to remove covered *t*-way sequences from the set of candidates during test sequence generation. After the generation finished, we consider the remaining uncovered *t*-way sequence candidates as invalid. The reasons why our solution works are as follows.

(1) Covered target sequences must be valid: According to our definition of valid sequence, and our previous research on constraint handling [9], all subsequences covered by a test sequence are valid. In other words, an invalid *t*-way sequence cannot be covered by any test sequence.

(2) Valid target sequences must be covered: For each starting phase, it ensures to cover at least one remaining *t*-way sequence candidate, until no such starting sequence can be created. So, before the break in Line 6, all valid *t*-way sequence candidates must have been covered by test sequences.

The second challenge of our generation approach is to create a starting sequence, i.e., a valid or extendable test sequence that covers at least one remaining *t*-way target sequence, within a reasonable time.

As indicated in Line 4, in order to ensure termination of test sequence generation, we create a starting test sequence that covers at least one target sequence in Π. Our solution is to adopt a BFS

(Breadth-First Search) strategy as in Fig. 4-12. Note that, validity and extensibility check can only guarantee to generate a complete test sequence, not for coverage, which may not cover any remaining target sequence.

```
Input: (a) A sequencing model M = (E, C), where E is a set of events and C is a set of constraints,
(b) a test strength t, and (c) a set of remaining target sequence candidates Π
Output: A starting test sequence Q
{
   // Initialize a queue of starting sequence candidates U
   let U be a queue consisting of all the event sequences of length t
   while (U is not empty) {
      remove the first sequence Q from U
      if (Q covers at least one target sequence in Π) {
         if (Q is valid or extendable)
            return Q
      }
      else if (Q is extendable) {
         for (each event e in E)
            append e to Q and add it to the end of U
      }
   }
   return null
}
```

Figure 4-12. Algorithm for creating a starting test sequence

4.5 Case Study

In this section, we apply our test sequence generation framework to the IEEE 11073-20601 protocol (Optimized Exchange Protocol) [10]. As a core component in the standards family of IEEE 11073, this protocol defines a communication model that allows PHDs (Personal Healthcare Devices) to exchange data with computing devices like mobile phones, set-top boxes, and personal computers.

4.5.1 Overview of the Protocol

In IEEE 11073, there are two types of devices, agent and manager devices. Agents are personal healthcare devices that are used to obtain measured health data from the user. Examples of agents include blood pressure monitors, weighing scales and blood glucose monitors. Managers manage and process the data collected by agents. Examples of managers include mobile phones, set-top boxes and PCs.

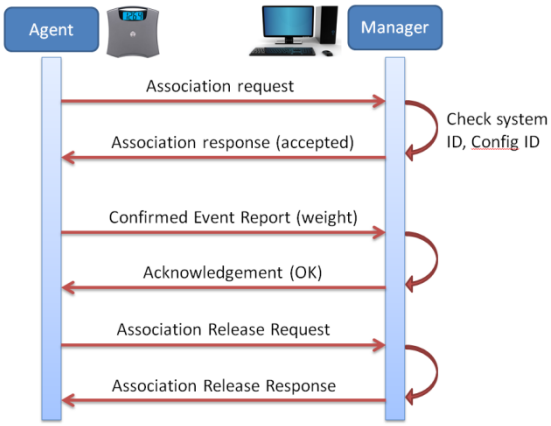

Figure 4-13. An example scenario of Data Exchange

We reuse an example scenario from our earlier work [8] to illustrate how an agent exchanges data with a manager, as shown in Fig. 4-13. In the scenario, the agent device is a weighting scale. It sends an *Association request* to the manager. The association request contains the weighting scale's configuration information, e.g., system ID, protocol version number. If the manager recognizes the agent, it accepts the association request and sends to the agent an *Association acceptance* message. At this point, the two devices are ready to exchange actual data. Next the agent sends measurement data, e.g., weight information, to the manager using a *Confirmed Event*

Report message. The manager successfully receives the Confirmed Event Report and sends back the acknowledgement. At the end of this scenario the agent requests to release the association with an *Association release request* message, and the manager releases the association and sends back to the agent an *Association release response* message.

In this case study, we identify sequencing constraints that the protocol imposes on the communication behavior. We specify these constraints using the notation developed in Section 4.3 and generate *t*-way test sequences that satisfy these constraints. These test sequences can be used to perform conformance testing of an implementation of the protocol, e.g., Antidote [11].

4.5.2 Sequencing Constraints

We identify constraints from the manager's perspective. Constraints can be similarly identified from the agent's perspective. In particular, as participants of the same protocol, agent and manager exhibit to a large extent symmetrical behavior, in terms that a send event on one side corresponds to a receive event on the other side.

The events on the manager side can be divided into three groups, based on their source and destination:

- Event beginning with REQ – There is a single request event, *REQ_assoc_rel*, sent from the application software interface, and it is triggered and handled inside the manager.

- Events beginning with Rx – These events are requests sent from the agent to the manager. They include *Rx_assoc_rel_req, Rx_assoc_rel_rsp, Rx_assoc_req, Rx_config_event_report_req*.

- Events beginning with Tx – These events are responses sent from the manager to the agent. They include *Tx_assoc_rel_req, Tx_assoc_rel_rsp, Tx_assoc_rsp_rejected, Tx_assoc_rsp_accepted, Tx_assoc_rsp_accepted_unknown_config, Tx_config_event_report_rsp_accepted_config, Tx_config_event_report_rsp_unsupported_config*.

Thus, there is a total of 12 events including 1 REQ, 4 Rx, and 7 Tx events for the manager. Note that we ignore the abort events which can happen anywhere, since we focus on positive testing.

Alternatively, the events can be divided into three groups, based on their functional areas.

- Events that establish association:

  Rx_assoc_req and its 3 possible responses *Tx_assoc_rsp_rejected, Tx_assoc_rsp_accepted, Tx_assoc_rsp_accepted_unknown_config*;

- Events that release association:

  Rx_assoc_rel_req and its response *Tx_assoc_rel_rsp*;

  REQ_assoc_rel, *Tx_assoc_rel_req* and its response *Rx_assoc_rel_rsp*;

- Events that check configuration:

  Rx_config_event_report_req and its 2 possible responses *Tx_config_event_report_rsp_accepted_config*, *Tx_config_event_report_rsp_unsupported_config*.

In our study, we first identify constraints from each functional group separately and then put them together, e.g., using nested expressions. The final constraints are shown in Fig. 4-14.

1. *Rx_assoc_req* ···* {all the events except *Rx_assoc_req*}
2. {all other events except the right three events} *··· {*Tx_assoc_rel_rsp, Rx_assoc_rel_rsp, Tx_assoc_rsp_rejected*}
3. *Rx_assoc_rel_req* − *Tx_assoc_rel_rsp*
4. *Rx_assoc_req* − {*Tx_assoc_rsp_rejected, Tx_assoc_rsp_accepted, Tx_assoc_rsp_accepted_unknown_config*}
5. *Rx_config_event_report_req* − {*Tx_config_event_report_rsp_accepted_config, Tx_config_event_report_rsp_unsupported_config*}
6. (*Tx_assoc_rel_req* ···* *Rx_assoc_rel_rsp*) && (*Tx_assoc_rel_req* ·~· {*Tx_assoc_rel_rsp, Rx_assoc_rel_rsp*} *··· *Rx_assoc_rel_rsp*)
7. {*Tx_assoc_rsp_accepted, Tx_assoc_rsp_accepted_unknown_config*} ·~· {*Tx_assoc_rel_rsp, Rx_assoc_rel_rsp*} ···* {*Rx_assoc_rel_req, REQ_assoc_rel*}
8. (*REQ_assoc_rel* − *Tx_assoc_rel_req*) && (*Tx_assoc_rel_req* *− {*Rx_assoc_rsp, Rx_assoc_rel_req*}) && (*Tx_assoc_rel_req* −* *Rx_assoc_rel_rsp*)
9. {*Tx_assoc_rsp_accepted_unknown_config, Tx_config_event_report_rsp_unsupported_config*} ·~· {*Rx_assoc_rel_req, REQ_assoc_rel*} ···* *Rx_config_event_report_req*

Figure 4-14. All 9 sequencing constraints of PHD manager model

Constraint 1. *Rx_assoc_req* is the first event that must happen before all other events. This event requests association to be established.

Constraint 2. *Tx_assoc_rel_rsp, Rx_assoc_rel_rsp, Tx_assoc_rsp_rejected* are the last events that must happen after all other events. These events indicate that association has been released or rejected.

Constraints 3-5. For convenience, we write "$e_1 -\!\!* e_2$ && $e_1 *\!\!- e_2$" in its abbreviated form "$e_1 - e_2$". Based on the protocol semantics, after the manager receives a request, it must immediately transmit an event as response.

Constraint 6. To maintain causal semantics, when a response event happens, its corresponding request event must happen before it. However, after a request event occurs, a response event may not always happen, e.g., due to disconnection or disassociation.

In the PHD protocol, after a request is transmitted, an event of its possible response may not be received when association has already been released by other events, which is indicated by events *Tx_assoc_rel_rsp* or *Rx_assoc_rel_rsp*. In other words, if these two events don't happen after *Tx_assoc_rel_req*, then *Rx_assoc_rel_rsp* must happen in some time.

Constraint 7. We have two request events *Rx_assoc_rel_req* and *REQ_assoc_rel* to release association from agent and manager side. These two events can only happen when the association is accepted and not yet released.

Constraint 8. When the manager triggers *REQ_assoc_rel*, it will immediately transmit a release request, and then busy wait until it receives either the release response or another release

request from the agent. The constraint restricts that the manager must not finish association release until the agent agrees.

Constraint 9. Similar to constraint 7, *Rx_config_event_report_req*, which receives a new configuration from the agent, can only happen after the previous configuration is checked to be unknown or unsupported and no release request has happened.

One benefit of our notation is that it allows incremental specification. That is, we do not require all the constraints be specified up front. Instead, we can begin with several constraints, generate test sequences that satisfy these constraints, and then check whether these sequences are as expected. If not, we can add more constraints. This can be repeated for multiple times until we capture all the constraints.

4.5.3 Test Sequence Generation Results

TABLE 4-2. RESULTS OF 2-WAY TEST SEQUENCE GENERATION

Rep cons	**Len cons**	**# of target seqs**	**Gen Time(sec)**	**# of test seqs**	**test seq length**		
					min	**avg**	**max**
≤ 1	≤ 6	36	3.9	7	2	4.6	6
≤ 1	≤ 7	45	26.7	9	2	5.1	7
≤ 1	≤ 8	45	155.5	9	2	5.1	7
≤ 2	≤ 6	61	13.1	15	4	5.7	6
≤ 2	≤ 7	79	157.1	16	4	6.4	7
≤ 2	≤ 8	105	1789.5	26	4	7.4	8
≤ 2	≤ 9	123	19802.9	24	4	8.2	9
≤ 2	≤ 10	135	206191.9	24	4	8.4	10

The experimental environment is set up as the following: OS: Windows 7 64bits, CPU: Intel Dual-Core i5 2.5GHz, Memory: 8 GB DDR3, SDK: Java SE 1.7.

We use the 9 sequencing constraints in Fig. 4-14 with different repetition and length constraints (as shown in the first two columns of TABLE 4-2) to generate test sequences that achieve 2-way sequence coverage.

TABLE 4-2 shows that the test generation time grows quickly as the maximum length of a test sequence increases. We believe our test generation algorithm has a lot of room for optimization, which will be explored in our future work. In fact, we have optimized our algorithm of *t*-way test sequence generation in Chapter 5, whose time complexity is greatly reduced.

Note that after we set the maximum repetition and maximum length constraint, test sequences may not grow up to the maximum length. For example, in TABLE 4-2, for the third experiment, where each event can only appear once and the length limit is 8, the maximum length of a test sequence we generate is 7. The reason is that sequencing and repetition constraints may interact to reduce the maximal length of a test sequence.

Also note that the number of target sequences increases as we relax the repetition and length constraints. Since the number of events is 12, the test strength is 2, the number of possible 2-way sequences is 12^2 = 144. Some of them cannot be covered due to repetition, length, and sequencing constraints.

4.6 Related Work

Combinatorial testing has been an active area of research [6]. However, most work has focused on t-way test data generation [12]. In this section, we focus our discussion on *t*-way sequence generation that supports constraints.

There exist many *t*-way test sequence generation approaches supporting constraints. However, some of them lack the capability to specify all possible constraints for real-life systems, while others require a low-level specification of constraints such as dependency graph or state transition diagram.

Kuhn et al. [4] presented an approach to generating *t*-way SCAs. Their approach requires each event to appear exactly once in a test sequence. Thus the length of each test sequence is fixed, which equals the number of events. It supports one type of constraint on sequence "*x*..*y*", which means that no test sequence should contain *x* and *y* in the given order. This is similar to our notation "$x \cdot\sim\cdot\ y$". This notation cannot specify constraints involving more than two events. For example, it cannot specify that some event must or never happen between two events. Furthermore, there are certain types of constraints between two events that cannot be specified by this notation. For example, consider the constraint in our notation, "$x \cdots{*}\ y$", meaning that if *y* happens, *x* must happen before *y*. This constraint cannot be specified using the notation in [4] to prevent sequence "*y*..*x*". This is because a test sequence in the form of "[… *x* … *y* … *x* … *y* …]" satisfies this constraint, but *x* and *y* appear in different orders in the same sequence.

Farchi et al. [13] developed an approach to generating test sets that satisfy ordered and unordered interaction coverage. Ordered restrictions can be considered as a type of sequencing constraints. For example, the ordered restriction excluding a case "Read.comesBefore(Open)" to prevent <Read, Open> from generation. This restriction is similar to the notation in [4], and thus has similar limitations as mentioned earlier.

Several approaches have been reported that use a graph model to represent system behavior from which *t*-way test sequences are generated. Wang et al. [14] presented a pairwise test sequence generation approach for web applications. Their approach is based on a graph model called navigation graph that captures the navigation structure of a web application. Rahman et al. [15] presented a test sequence generation approach using simulated annealing. Their approach is based on a state transition diagram that models the system behavior. Yu et al. [7][8] presented several algorithms that generate *t*-way test sequences from LTS models. In these approaches, sequencing constraints are implicitly encoded in the graph model. Compared to our notation, the graph models used in these approaches are at a lower level of abstraction and require a lot of operational details that may not be readily available in practice.

Kruse et al. [16] suggested that temporal logic formulas, e.g., Linear Temporal Logic (LTL) [17], Computational Tree Logic (CTL) [18], and modal μ-calculus [19], can be used to express sequencing constraints. They used LTL for dependency rules (i.e., sequencing constraints) and CTL for generation rules (i.e., strength *t*, repetition and length constraints). Temporal logic

formulas are powerful in terms of the different types of property they could be used to express. However, these notations have a complex semantic model, and have found limited use in practice. For example, both LTL and CTL have a state-based semantic model. In theory, any state-based property can be specified using events, and vice versa. However, the notion of state is more difficult to grasp than that of event. This is because unlike events, states are not directly represented in a test sequence. Thus, in order to specify sequencing constraints, events must be translated into states. This translation can be difficult due to the fact that states can be defined at different levels of abstraction and thus the mapping between states and events may not be a simple one-to-one relation.

Dwyer et al. [20] developed a system of property specification patterns to specify properties that are commonly encountered in practice. Our work is different in that we define a minimal set of basic operators, each of which captures a fundamental relationship between events. Complex properties can be specified using these basic operators. The work in [20] is complementary with ours in that similar patterns can also be identified to facilitate the use of our notation in practice.

4.7 Conclusion and Future Work

There seems to be a significant amount of interests on *t*-way sequence testing in both academia and industry. However, progress is still lacking. In this paper we present an approach to handling sequencing constraints, which we believe is a key technical challenge in *t*-way test sequence generation but has not been adequately addressed. Our approach consists of an

event-oriented notation for expressing sequencing constraints and a greedy algorithm for generating test sequences that achieve *t*-way coverage while ensuring that all the constraints are satisfied. We applied our approach to a real-life communication protocol. Our experience suggests that our notation is more intuitive to use and can capture important sequencing constraints for this protocol. However, our test generation algorithm seems to be time consuming. This work is part of our larger and ongoing effort to make *t*-way sequencing testing practically useful.

In the future, we will continue our work in the following major directions. First, we want to optimize the performance of our test sequence generation algorithms. For example, there seems to be quite some redundant computations in the generation process. We plan to explore ways to reduce such redundancy, e.g., by saving intermediate results. Second, we want to develop an algorithm to perform consistency check on constraints specified by the user. This is necessary because the user may specify constraints that contradict with each other. This consistency check can reject contradictory constraints prior to test generation and can also provide feedback to the user in terms of how to make corrections. Finally, we want to investigate the formal properties of our notation for sequencing constraints, in terms of what kind of constraints our notation can or cannot express. In particular, we want to check the possible equivalence relation between our notation and other notations such as LTS and LTL. For example, is it true that any properties that can be expressed using LTS or LTL can be expressed using our notation and vice versa?

4.8 Acknowledgment

This research is partly supported by two research grants (70NANB15H199, 70NANB18H207) from National Institute of Standards and Technology.

Disclaimer: We identify certain software products in this document, but such identification does not imply recommendation by the US National Institute of Standards and Technology or other agencies of the US government, nor does it imply that the products identified are necessarily the best available for the purpose.

4.9 References

[1] A. Canny. “Interactive system testing: beyond GUI testing.” In Proceedings of the 2018 ACM SIGCHI Symposium on Engineering Interactive Computing Systems, p. 18.

[2] R.C. Bryce, S. Sampath, and A.M. Memon. “Developing a single model and test prioritization strategies for event-driven software.” IEEE Transactions on Software Engineering 37, no. 1 (2011): 48-64.

[3] D.E. Simos, J. Bozic, B. Garn, M. Leithner, F. Duan, K. Kleine, Y. Lei, and F. Wotawa. “Testing TLS using planning-based combinatorial methods and execution framework.” Software Quality Journal (2018): 1-27.

[4] D.R. Kuhn, J.M. Higdon, J.F. Lawrence, R.N. Kacker, and Y. Lei. “Combinatorial methods for event sequence testing.” In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 601-609.

[5] F. J. Daniels, and K. C. Tai. “Measuring the effectiveness of method test sequences derived from sequencing constraints.” In Proceedings of the 1999 Technology of Object-Oriented Languages and Systems, pp. 74-83.

[6] C. Yilmaz, S. Fouche, M.B. Cohen, A. Porter, G. Demiroz, and U. Koc. “Moving forward with combinatorial interaction testing.” Computer 47, no. 2 (2014): 37-45.

[7] L. Yu, Y. Lei, R.N. 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 (ICECCS), pp. 220-229.

[8] L. Yu, Y. Lei, R.N. Kacker, D.R. Kuhn, R.D. Sriram, and K. Brady. “A general conformance testing framework for IEEE 11073 PHD's communication model.” In sixth International

Conference on Pervasive Technologies Related to Assistive Environments (PETRA 2013), p. 12.

[9] L. Yu, Y. Lei, M. Nourozborazjany, R.N. Kacker, and D. R. Kuhn. “An efficient algorithm for constraint handling in combinatorial test generation.” In 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), pp. 242-251.

[10] J.H. Lim, C. Park, S.J. Park, and K.C. Lee, “ISO/IEEE 11073 PHD message generation toolkit to standardize healthcare device.” In 2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society. pp. 1161-1164.

[11] Antidote IEEE 11073-20601 stack library. [Online]. http://oss.signove.com/index.php/Antidote:_IEEE_11073-20601_stack

[12] K. Kleine, and D.E. Simos. “An efficient design and implementation of the In-Parameter-Order algorithm.” Mathematics in Computer Science 12, no. 1 (2018): 51-67.

[13] E. Farchi, I. Segall, R. Tzoref-Brill, and A. Zlotnick. “Combinatorial testing with order requirements.” In 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 118-127.

[14] W. Wang, S. Sampath, Y. Lei, and R.N. Kacker. “An interaction-based test sequence generation approach for testing web applications.” In Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium (HASE), pp. 209-218

[15] M. Rahman, R.R. Othman, R.B. Ahmad, and M.M. Rahman. “Event driven input sequence t-way test strategy using simulated annealing.” In 2014 5th International Conference on Intelligent Systems, Modelling and Simulation (ISMS), pp. 663-667.

[16] P.M. Kruse, and J. Wegener. “Test sequence generation from classification trees.” In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 539-548.

[17] A. Pnueli. “The temporal logic of programs.” In 1977 18th Annual Symposium on Foundations of Computer Science, pp. 46-57.

[18] E.M. Clarke, and E.A. Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logic.” In 1981 Workshop on Logic of Programs, pp. 52-71.

[19] D. Kozen. “Results on the propositional μ-calculus.” Theoretical computer science 27, no. 3 (1983): 333-354.

[20] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. “Patterns in property specifications for finite-state verification.” In Proceedings of 1999 21st International Conference on Software Engineering (ICSE), pp. 411-420.

T-WAY TEST SEQUENCE GENERATION USING AN EVENT-ORIENTED NOTATION

Feng Duan, Xiaolei Ren, Yu Lei, Raghu N. Kacker, and D. Richard Kuhn, “T-way Test Sequence Generation using an Event-Oriented Notation.” Submitted to the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021.

CHAPTER 5.

T-way Test Sequence Generation using an Event-Oriented Notation

Abstract—In this paper we design an event-oriented notation for specifying sequencing constraints and develop a t-way test sequence generation algorithm for handling constraints specified using this notation. To efficiently solve constraints, we translate our notation to Deterministic Finite Automaton (DFA) through operations on automata and use DFA to check sequence validity and extensibility during test generation. Then, we report a case study in which our constraint notation and generation algorithm are applied to a real-life communication protocol. Our experience indicates that our notation is intuitive to use and allows us to express important sequencing constraints for the protocol. Finally, we present how to analyze the generation results to find a local optimal set of test sequences that consists of the least number of sequences and has minimum sum of sequence lengths.

Keywords—*Sequential testing; Test sequence generation; Sequencing constraint; T-way sequence coverage; Event-based testing; Combinatorial testing*

5.1 Introduction

Many systems, e.g., interactive systems [1], event-driven systems [2] and communication protocols [3], exhibit sequencing behavior, where a sequence of events is exercised during each execution and the order in which the events occur could significantly affect the system behavior.

To test these systems, we need to generate test sequences, in addition to test data as parameter values.

As introduced in [32], combinatorial testing has been widely accepted as a way of detecting *t*-way interaction failures as two or more input values interact to cause the program to reach an incorrect result. The interaction level *t* is often referred to as *strength*. This concept of *t*-way testing has then been expanded from input values to event orders, called *t*-way sequence testing [4].

T-way sequence testing applies the notion of *t*-way coverage to test sequence generation. Informally, given any *t* events, if they could be exercised in a given order, there must exist at least one test sequence in which these events are exercised in this order. Doing so allows us to test all possible interactions among any *t* events. Thus, *t*-way sequence testing can expose faults that are caused by interactions among no more than *t* events.

One important problem in *t*-way sequence testing is dealing with *sequencing constraints* [5], i.e., restrictions on the order of events that must be satisfied for a test sequence to be valid. The technical challenge is two-fold. First, a notation is needed to specify sequencing constraints. This notation must be easy to use and have the power to express commonly encountered constraints. Second, a test generation algorithm must be developed to handle sequencing constraints. Compared to constraints on data values, sequencing constraints can be more difficult to handle. This is because the space that needs to be searched in the evaluation process for sequencing constraints can be much larger due to the extra dimension, i.e., order of events.

Recent years have seen significant progress on *t*-way test data generation, but not on *t*-way test sequence generation [6]. Kuhn et al. [4] presented an approach to generate Sequence Covering Arrays (SCAs) for testing special systems. Their approach requires each event occurs exactly once in each test sequence, and only supports one type of constraint. Yu et al. [7][8] presented another approach to *t*-way test sequence generation based on a Labeled Transition System (LTS) model that captures system behavior. An LTS model is similar to a Finite State Machine (FSM) where the sequencing constraints are implicitly encoded in the state transitions. Bombarda et al. [29] [30] presented methods for sequential test generation by using FSM. However, as an operational model, LTS is at a very low level of abstraction and requires a lot of details on how a system operates in terms of states and transitions. As a result, LTS models are seldom available in practice.

In this paper we design a notation for expressing commonly used sequencing constraints and develop a *t*-way test sequence generation algorithm for handling constraints expressed using this notation. Our notation adopts an event-oriented framework. It defines a small set of operators that capture fundamental order restrictions that could happen between two events. These operators can be nested, if necessary, to specify the sequencing behavior among multiple events. Our notation is at a higher level of abstraction than an operational model such as LTS. Also, we believe that since we deal with sequences of events, an event-oriented notation is more intuitive than a state-machine-based notation.

Our test sequence generation algorithm employs a greedy strategy in which each test sequence is generated such that a maximal number of *t*-way target sequences can be covered in the sequence. A *t*-way target sequence is a sequence of *t* events that could be covered in the given order, i.e., the order in which they appear in the sequence. Each test sequence is generated in two phases, including the starting phase and the extension phase. In the starting phase, we generate a starting sequence that is guaranteed to cover at least one target sequence. In the extension phase, we keep extending the test sequence until no extension is possible. At each extension, we append to the test sequence an event that covers the most *t*-way target sequences that are yet to be covered.

We report a case study in which we applied our approach to a communication protocol, i.e., IEEE 11073-20601 [10]. This protocol is used to exchange data between Personal Health Devices (PHDs), e.g., blood pressure monitors or weighing scales, and computing devices, e.g., mobile phones or desktop computers. We identify a set of 9 sequencing constraints and write them using our notation. We generate test sequence sets that achieve 2-way and 3-way sequence coverage while satisfying all the sequencing constraints. Our experience indicates that our notation is intuitive to use and allows us to express important sequencing constraints for this protocol. The generation results are analyzed to find a local optimal set of test sequences having the least number of sequences and the minimum sum of sequence lengths.

We focus on positive testing in this paper. That is, we generate test sequences that satisfy all the sequencing constraints. Note that sequencing constraints are also useful for negative testing where test sequences could be generated to violate constraints in a systematic manner.

The remainder of the paper is organized as follows. Section 5.2 explains our basic idea with a motivating example. Section 5.3 presents the syntax and semantics of our notation to specify sequencing constraints. We also introduce two other types of constraints, i.e., repetition and length constraints, which are used to control test sequence length. Section 5.4 first presents the definitions of *t*-way target and test sequences, second explains the translation of sequencing constraints to Deterministic Finite Automaton (DFA), and then describes our test generation algorithm that handles constraints by using DFA. Section 5.5 presents the results of our case study on the PHD protocol, including the sequencing constraints identified and some statistics on the test sequences generated. Section 5.6 discusses related work on *t*-way test sequence generation. Section 5.7 provides concluding remarks and our plan for future work.

5.2 Motivating Example

In this section, we use File API as a motivating example to show the basic idea of our work, including how to model the sequencing behavior of a system in terms of events and constraints, and how to generate a set of test sequences for *t*-way sequence coverage.

5.2.1 Model Sequencing Behavior

A *sequencing model* M = <E, S> consists of two components: (1) E: a set of events that could be exercised in a system execution; (2) S: a set of constraints that restrict the occurrences of these events in a system execution.

There are four major file operations in File API, including *open*, *close*, *read*, and *write*. In the sequencing model, each of these operations is modeled as an event. Thus, E = {*open*, *close*, *read*, *write*}.

Based on the semantics of file operations, the following constraints can be identified in terms of the order in which these events could be exercised. These constraints are referred to as sequencing constraints.

(1) A file must be closed after *open*.

(2) A file must be open just before *read*, *write*, or *close*.

We introduce an event-oriented notation to model the above constraints. To model the first constraint, we specify that an occurrence of event *close* must exist after any occurrence of event *open*. To model the second constraint, we specify that an occurrence of event *open* must exist before any occurrence of events *read*/*write*/*close* and there shall be no *close* in between.

In addition to sequencing constraints, we introduce two other types of constraints, namely repetition and length constraints, to control the length of a test sequence. A repetition constraint specifies the number of times a certain event could be repeated in a test sequence. For example,

we may specify that *open*/*close* could only occur once in a test sequence. A length constraint specifies the maximum length of a test sequence.

In Section 5.3, we introduce a formal notation to specify the three types of constraints in detail.

5.2.2 Generate Test Sequence

After a sequencing model is specified, a test sequence set can be generated to achieve *t*-way sequence coverage. Recall that *t*-way sequence coverage requires that every *t*-way (target) sequence, i.e., every sequence of *t* events that could be exercised in the given order, consecutively or not, be exercised so by at least one test sequence.

For example, for File API, the set of 2-way sequences is {<*open*, *open*>, <*open*, *close*>, ..., <*read*, *write*>, ..., <*write*, *write*>} (the set size = 4^2 = 16). Note that the existence of constraints may make some of these sequences *uncoverable*, i.e., they cannot be covered by any test sequence that satisfies all the constraints. In this paper, we use <e_1, e_2, ...> to represent target sequences, and use [e_1, e_2, ...] to represent test sequences. A 2-way target sequence <e_1, e_2> is covered by a test sequence in the form of [..., e_1, ..., e_2, ...].

If we specify some repetition constraints such that no event could be repeated in a test sequence, in addition to the two sequencing constraints mentioned earlier, there is a total of seven 2-way target sequences {<*open*, close>, <*open*, *read*>, <*open*, *write*>, <*read*, *close*>, <*read*, *write*>, <*write*, *close*>, <*write*, *read*>}. A greedy algorithm can be used to generate a 2-way test

sequence set such that all the test sequences satisfy all the constraints, and every target sequence is covered by at least one test sequence.

The details of the greedy algorithm are presented in Section 5.4. The following example illustrates the basic idea of the algorithm:

1. We first construct a starting sequence that covers at least one target sequence. We begin with an empty test sequence. The only possible event that could be added is *open* due to its 2nd sequencing constraint. The resulting sequence is [*open*]. We further extend [*open*]. There are three possible choices: [*open, read*]/[*open, write*]/[*open, close*]. Note that [*open, open*] is not allowed due to the repetition constraint. All of the three choices cover one target sequence. We choose [*open, read*] as our starting sequence.

2. Now we try to extend the starting sequence. We append to the sequence one event at a time. We first append *write*, followed by *close,* as these two events allow the most target sequences to be covered. The resulting sequence is [*open, read, write, close*], which cannot be further extended.

3. We repeat the above two steps to generate additional test sequences until all the remaining target sequences are covered.

The final 2-way test sequence set consists of two test sequences {[*open, read, write, close*], [*open, write, read, close*]}. These two sequences satisfy all the constraints and cover all of the seven 2-way target sequences.

5.3 Notation for Constraint Specification

Our notation supports three types of constraints, including repetition, length, and sequencing constraints. Recall that repetition and length constraints are used to control the length of a test sequence.

First, we introduce the syntax and semantics of repetition and length constraints:

- A repetition constraint for single event *e* is in the form of "Rep(*e*) ≤ *r*", denoting that, in a test sequence, an event *e* could occur no more than *r* times. A default repetition constraint for all events can be specified in the form of "Rep ≤ *r*", denoting that no event could occur for more than *r* times. The default constraint can be overridden by an event-specific constraint.
- A length constraint is in the form of "Len ≤ *max*", denoting that, the length of any test sequence should be no more than *max*.

The rest of this section is focused on the syntax and semantics of sequencing constraints.

5.3.1 Syntax of Sequencing Constraints

The syntax of sequencing constraints is specified in BNF (Backus–Naur form) as shown in Fig. 5-1.

```
<constraint> ::= <constraint> || <constraint>
      | <constraint> && <constraint>
      | <constraint> . <constraint>
      | !<constraint>
      | <constraint> <immediate operator> <constraint>
      | <always operator> <constraint>
      | (<constraint>)
      | <event set>
<event set> ::= {<event list>} | <event>
<event list> ::= <event> , <event list> | <event>
<event> ::= (<letter> | <digit>) (<letter> | <digit>)*
<letter> ::= "A"-"Z" | "a"-"z" | "_" | "/"    //Labels need "/" in Mealy Machine
<digit> ::= "0"-"9"
<always operator> ::= "_"    //Need whitespace to split due to labels allow "_"
<immediate operator> ::= "+-" | "-+" | "~"
```

Figure 5-1. BNF of sequencing constraints

Inside the BNF, its event identifiers are basically from the set of events that could be exercised in System Under Test (SUT). There are two types of operators in high-level: *Regular* and *Sequencing operators*, while Sequencing operators can be furtherly divided into two sub-types: *always* and *immediate*.

Always operator and three immediate operators are referred to as *Sequencing operators* since they are defined to construct sequencing constraint with operands starting from events. The semantics of Sequencing operators are defined in following parts which indicate that any sequencing constraint can be translated to an equivalent automaton.

The four operators *NOT* "!", *CONCAT* ".", *AND* "&&", *OR* "||" are referred to as *Regular operators* since they are defined and used as regular operations on automata [23][24]. As introduced in the BNF, each operand can be a constraint which equals an automaton, Regular operators on constraints are mapping to the regular operations on automata as follows: (1) *NOT* "!" = the *Complement (Negation)* of the automaton equivalent to constraint; (2) *CONCAT* "." = the *Concatenation* of left automaton and then right automaton; (3) *AND* "&&" = the *Intersection* of two automata; (4) *OR* "||" = the *Union* of two automata. The details of translation from constraint to automaton can be found in Section 5.4.

Note that the precedence of the operators is defined from highest to lowest as follows: PAREN "()", always operator, immediate operators, NOT "!", CONCAT ".", AND "&&", OR "||".

A sequencing constraint can be derived from this syntax as shown in Fig. 5-2.

<constraint>

=> <constraint> <immediate operator> <constraint>

=> (<constraint>) <immediate operator> <event set>

=> (<constraint> && < constraint>) <immediate operator> <event set>

=> (<always operator> <constraint> && <constraint> <immediate operator> <constraint>) <immediate operator> <event set>

=> (<always operator> <event set> && <event set> <immediate operator> <always operator> <event set>) <immediate operator> <event set>

=> (<always operator> <event> && <event> <immediate operator> <always operator> <event>) <immediate operator> <event set>

=> (_ *open* && *open* ~ _ *close*) -+ {*read, write, close*}

Figure 5-2. Derivation of a sequencing constraint from BNF

5.3.2 Basic Semantics

TABLE 5-1. INFORMAL EXPLANATION OF SEQUENCING OPERATORS

Sequencing operator	Explanation considering operands are single event
_ e_1 (or $\underline{e_1}$)	e_1 **always happens** at least once
e_1 +- e_2	If e_1 happens, then e_2 **must happen immediately after** e_1
e_1 -+ e_2	If e_2 happens, then e_1 **must happen immediately before** e_2 (i.e., e_2 **can only happen immediately after** e_1 happens)
e_1 ~ e_2	e_2 **never happens immediately after** e_1 (or e_1 **never happens immediately before** e_2)

TABLE 5-1 lists all the Sequencing operators in our notation and provides an informal explanation of each operator considering its operands are single event. We make the following notes about the symbols used to represent the operators:

(a) UNDERLINE "_" : This indicates that the event always happens at least once. (In this paper, this operator may be adopted in two forms, i.e., "_ e_1" or "$\underline{e_1}$")

(b) DASH "-" : This indicates the left event happens immediately before the right event, or in other words, the right event happens immediately after the left event.

(c) WAVE "~" : This indicates the left event never happens immediately before the right event, or in other words, the right event never happens immediately after the left event.

(d) POSITIVE "+" : This indicates that, when "+" appears left (or right) in sequencing operators, the constraint applies only if the left (or right) event happens.

Recall that in BNF we allow sequencing operators to be applied to a set of events. For example, "e_1 +- {e_2, e_3}" denotes that, if event e_1 happens, an event in set {e_2, e_3} must happen immediately after it. Note that "e_1 +- {e_2, e_3}" = "e_1 +- (e_2 || e_3)" which nests a Regular operator "||" to denote the same constraint.

The basic semantics of Sequencing operators on single events or event sets are represented by automata as below, in form of *complete* DFA determinized from Nondeterministic Finite Automaton (NFA) [25][26]. (**complete**: all state-label transitions have been drawn. If the destination of some transition is Error state, then Error state with self-loop transition via labels of all events should also be introduced.)

We adopt Regular Expression (Regex) [21][22] to express automaton. Note that in each automaton, ∀e indicates arbitrary event in the event set of SUT; ¬e_1 indicates any event in complementary set of event set e_1; $e_1 \vee e_2$ indicates any event in either set e_1 or e_2; $e_1 \wedge e_2$ indicates any event in both set e_1 and e_2; e_1/e_2 indicates any event in set e_1 but not in e_2.

5.3.2.1 Always Sequencing Operator

(1) _ e_1

According to the explanation of always operator, "_ e_1" (a.k.a. "$\underline{e_1}$") = NFA "e*e_1e*" (representing in form of Regex) = DFA "(¬e_1)*e_1e*" as shown in Fig. 5-3. Note that "e*",

abbreviated from "(∀e)*", represents a Regex using Kleene Star on arbitrary event in the event set of SUT which accepts any possible event sequence.

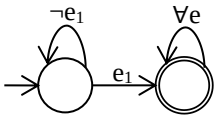

Figure 5-3. Semantics of "_ e_1"

Fig. 5-3 shows that, given an input sequence, if an occurrence of any event in set e_1 exists, the sequence should be accepted. Otherwise, it is rejected.

It is easy to find "!(_ e_1)" = DFA "(¬e_1)*" by getting the Complement (Negation) of the complete DFA of "_ e_1", i.e., the set of forbidden sequences which violate "_ e_1" is equivalent to the language of Regex "(¬e_1)*". Note that "¬e_1" ≠ "!e_1", since "¬e_1" is the complementary event set of e_1, "!e_1" is the Complement of automaton "e_1" that "!e_1" = DFA "(¬e_1)e* | e_1(∀e)e*" which accepts any sequence except single event in e_1.

5.3.2.2 Immediate Sequencing Operators

The DFA of each immediate operator is derived from the Complement of a negative NFA which accepts all its forbidden sequences.

(2) e_1 +- e_2

NFA "e*e_1!(e_2e*)" accepts all forbidden sequences that violate constraint "e_1 +- e_2", since each sequence of its language has an occurrence of some event in e_1 whose immediately rear

subsequence does not begin with any event in e_2. Following five steps as below, we can get the DFA of "e_1 +- e_2" as shown in Fig. 5-4:

Step 1. Determinize NFA "e^*e_1" by constructing subsets of its states [25][26]. The result is "e^*e_1" = DFA "$(\neg e_1)^*e_1e_1^*(\neg e_1(\neg e_1)^*e_1e_1^*)^*$".

Step 2. Get the DFA of "$!(e_2e^*)$" as the Complement of the complete DFA of "e_2e^*" by reversing the accept status of all states in complete DFA. (In this paper, we do not remove the dead states which cannot reach accept states but merge them into an Error state in order to make all DFAs complete.) Thus, "$!(e_2e^*)$" = DFA "$\varepsilon \mid \neg e_2e^*$". ($\varepsilon$ represents empty sequence which does not require any event to happen).

Step 3. Concatenate the DFA of "e^*e_1" with the DFA of "$!(e_2e^*)$" to get a NFA "$(\neg e_1)^*e_1e_1^*(\neg e_1(\neg e_1)^*e_1e_1^*)^*(\varepsilon \mid \neg e_2e^*)$". Note that the possibility "$e_1 \wedge e_2$ may not be empty" should be considered for determinizing this NFA.

Step 4. Divide the whole event set of SUT into four individual sets e_1/e_2, e_2/e_1, $e_1 \wedge e_2$, $\neg(e_1 \vee e_2)$ which have none overlapped event. Use these four individual event sets for constructing subsets of states to determinize the above NFA of "$e^*e_1!(e_2e^*)$" into a complete DFA.

Step 5. Reverse the accept status of all states in the complete DFA of "$e^*e_1!(e_2e^*)$" to get its Complement. The result is "e_1 +- e_2" = NFA "$!(\ e^*e_1!(e_2e^*)\ )$" = DFA "$(\ \neg e_1 \mid e_1(e_1 \wedge e_2)^*(e_2/e_1)\ )^*$".

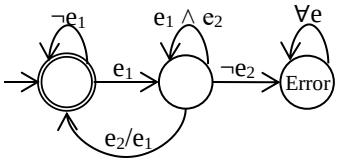

Figure 5-4. Semantics of "e_1 +- e_2"

Fig. 5-4 shows that, given an input sequence, if every occurrence of any event in set e_1 is immediately followed by an occurrence of any event in set e_2, the sequence should be accepted. Otherwise, it is rejected.

Note that event sets e_1 and e_2 may have some overlapped event, i.e., $e_1 \wedge e_2$ may not be empty. If so, the semantics of notation should follow its DFA but not the informal explanation, since the DFA has no ambiguity.

(3) e_1 -+ e_2

NFA "!(e*e_1)e_2e*" accepts all forbidden sequences that violate constraint "e_1 -+ e_2", since each sequence of its language has an occurrence of some event in e_2 whose immediately front subsequence does not end with any event in e_1. Thus, "e_1 -+ e_2" = NFA "!(!(e*e_1)e_2e*)" = DFA "!((¬(e_1∨e_2) | (e_1/e_2)e_1*¬e_1))*e_2e*)" = DFA "(¬(e_1∨e_2)|(e_1/e_2)e_1*¬e_1)* | (¬(e_1∨e_2))*(e_1/e_2)e_1*(¬e_1(¬(e_1∨e_2))*(e_1/e_2)e_1*)*".

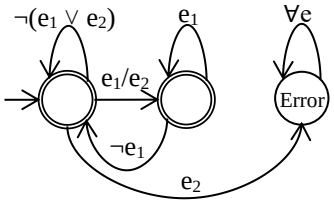

Figure 5-5. Semantics of "e_1 -+ e_2"

Fig. 5-5 shows that, given an input sequence, if any occurrence of any event in set e_2 exists that is NOT immediately led by an occurrence of any event in set e_1, the sequence should be rejected. Otherwise, it is accepted.

(4) $e_1 \sim e_2$

NFA "$e^*e_1e_2e^*$" accepts all forbidden sequences that violate constraint "$e_1 \sim e_2$", since each sequence of its language has an occurrence of some event in e_1 whose next event is in e_2. Thus, "$e_1 \sim e_2$" = NFA "!($e^*e_1e_2e^*$)" = DFA "!($(\neg e_1)^*e_1(e_1/e_2)^*(\neg(e_1 \vee e_2)(\neg e_1)^*e_1(e_1/e_2)^*)^*e_2e^*$)" = DFA "$(\neg e_1|e_1(e_1/e_2)^*\neg(e_1 \vee e_2))^* \mid (\neg e_1)^*e_1(e_1/e_2)^*(\neg(e_1 \vee e_2)(\neg e_1)^*e_1(e_1/e_2)^*)^*$".

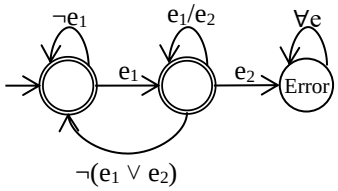

Figure 5-6. Semantics of "$e_1 \sim e_2$"

Fig. 5-6 shows that, given an input sequence, if any occurrence of any event in set e_2 exists immediately after an occurrence of any event in set e_1, the sequence should be rejected. Otherwise, it is accepted.

5.3.3 Nesting Feature

As we mentioned in BNF, always and immediate sequencing operators support their operands to be not only event set but also another sequencing constraint. We call this feature of our notation as Nesting Feature. Nesting Feature is introduced to support complex constraints that users may

provide. If we consider event e_1 to be an automaton B accepting e_1, event e_2 to be an automaton C accepting e_2, then we can easily understand Nesting Feature which supports operands to be constraints.

5.3.3.1 Semantics of Sequencing Operators on Constraints

(1) _ B

"_ B" (a.k.a. "B") = NFA "e*Be*".

The semantics is that a valid test sequence **always** contains at least one subsequence satisfying the constraint B (i.e., accepted by the automaton B). Note that we use B to represent both a constraint and its equivalent automaton.

(2) B +- C

"B +- C" = NFA "!(e*B!(Ce*))".

The semantics is that, given a valid test sequence, if any subsequence satisfies B, then there **must** exist at least one subsequence **immediately after** it which satisfies C.

(3) B -+ C

"B -+ C" = NFA "!(!(e*B)Ce*)".

The semantics is that, given a valid test sequence, if any subsequence satisfies C, then there **must** exist at least one subsequence **immediately before** it which satisfies B.

(4) B ~ C

“B ~ C” = NFA “!(e*BCe*)”.

The semantics is that, given a valid test sequence, if any subsequence satisfies B, then there **never** exists any subsequence **immediately after** it which satisfies C.

Note that there are some equivalent nested forms of Regular and Sequencing operators, such as “!(B ~ C)” = “_(B . C)” both of which are equivalent to NFA “e*BCe*”.

5.3.3.2 Nested Constraints for Real-life System

The Regular operators may also be nested inside constraint, whose semantics can be translated to regular operations on automata. The details of translation from constraint (nested with Regular and Sequencing operators) to automaton is discussed in Section 5.4.

Nested notation can be used to represent relationships not only between two sets of events, but also between two subsequences of events. In real-life cases, three types of meaningful relationships can be usually derived:

1. Cause-Effect relationship: “Cause +- Effect”, i.e., Cause must result in Effect. For example, “*open* +- *close*” indicates “an open file must be closed in time”.

2. Prerequisite-Action relationship: “Prerequisite -+ Action”, i.e., Action has to require Prerequisite. For example, “(*open* && *open* ~ *close*) -+ {*read*, *write*, *close*}” indicates “*read*/*write*/*close* file operations require the file must be open”.

3. Behavior-Exclusion relationship: “Behavior ~ Exclusion”, i.e., mutual exclusion relationship between a Behavior and its Exclusion in temporal logic. For example, “(*close* && *close* ~ *open*) ~ {*read, write, close*}” indicates “*read*/*write*/*close* file operations cannot work on a closed file”.

The first two nested notation in above example (while the third is redundant) exactly represent the sequencing constraints identified in Section 5.2. The set of two can be translated to DFA “(*open*(*open*|*read*|*write*)**close*)*” as the FSM of File API model.

In a word, Nesting Feature allows Regular and Sequencing operators being nested to represent complex constraints from real-life SUTs. The benefit of our Nesting Feature is that basic notation can be iteratively nested, until it represents an exact sequencing constraint among multiple events.

5.3.3.3 Special Semantics in Abbreviated Forms

For convenience, we allow user to write some special semantics in their abbreviated forms, in order to improve the readability of nested constraints.

TABLE 5-2. ABBREVIATED FORMS OF SPECIAL NESTED CONSTRAINTS

Abbr. form	Special nested constraint
B +... C	B +- C
B ...+ C	B -+ C
B .~. C	B ~ C (or equivalent form B ~ C)
B ... C	B +- (C && B ~ C) && (B && B ~ C) -+ C

According to the entries in TABLE 5-2, it is easy to informally explain the special semantics considering their operands are single event:

- "e_1 +... e_2" = "e_1 +- $\underline{e_2}$" = NFA "!(e*e_1!(e*e_2e*))", i.e., if e_1 happens, then e_2 **must happen (not necessarily immediately) after** e_1.
- "e_1 ...+ e_2" = "$\underline{e_1}$ -+ e_2" = NFA "!(!(e*e_1e*)e_2e*)", i.e., if e_2 happens, then e_1 **must happen (not necessarily immediately) before** e_2.
- "e_1 .~. e_2" = "e_1 ~ $\underline{e_2}$" (= "$\underline{e_1}$ ~ e_2") = NFA "!(e*e_1e*e_2e*)", i.e., e_2 **never happens after** e_1 (or e_1 **never happens before** e_2).
- "e_1 ... e_2" = "e_1 +- ($\underline{e_2}$ && $\underline{e_1}$ ~ e_2) && ($\underline{e_1}$ && e_1 ~ $\underline{e_2}$) -+ e_2" = DFA "(¬(e_1∨e_2) | e_1(¬(e_1∨e_2))*e_2)*", i.e., e_1 **must pair with** e_2 (not necessarily immediately, but requires neither e_1 nor e_2 occurs between them). Note that ¬(e_1∨e_2) indicates any event neither e_1 nor e_2.

5.3.4 Comparison of Our Notation with Automaton

As mentioned above and detailed in Section 5.4, the set of sequencing constraints from SUT can be translated to a DFA equivalent to its FSM. The reason why we specify constraints other than directly use FSM is as follows. Comparing to a whole FSM which consists of all relationships among events, one benefit of specifying multiple constraints by our notation is that when an event sequence is invalid for SUT (both FSM and the set of constraints can detect), we

can figure out which constraints it violates. Another benefit of our notation is that it allows incremental specification which is introduced in Section 5.5.

One more consideration is that, while each constraint expressed using our notation can be translated to a DFA, why we represent each constraint by our notation other than directly use DFA? The reason is that our notation is more user-friendly since it only requires the partial events explicitly mentioned in a constraint, while DFA requires to consider all the events in SUT. During the formulation of each constraint, our notation hides those implicit events, which helps user to focus on the relationship among explicit events. Note that those implicit events still play roles in solutions of this constraint, i.e., their relationships are implied in our notation and become explicit after translation to DFA for solving.

5.4 Approach

5.4.1 Basic Concepts

Informally, a target sequence is a sequence of events that needs to be covered. A test sequence is a sequence of events that can be executed by the subject system. Target sequences are covered by test sequences to satisfy *t*-way sequence coverage. In other words, target sequences are test requirements, i.e., entities that must be covered to achieve *t*-way sequence coverage, while test sequences are test cases that cover test requirements. T-way sequence coverage requires that for every target sequence of *t* events, if they could be exercised in the given order, they must be exercised so in at least one test sequence.

In the following, we formalize these concepts from the perspective of test sequence generation, without considering the semantics of the subject system. Let M = <E, S> be the sequencing model of the subject system.

Definition 1. A sequence Q of events is *valid* if it satisfies all the constraints in S; otherwise, it is invalid.

In this paper we focus on positive testing. Thus, every test sequence must be a valid sequence. Also, every valid sequence can be used as a test sequence.

Definition 2. A sequence Q of events is *extendable* if it is a proper prefix of another sequence of events that is valid.

Note that an extendable sequence itself may or may not be valid.

Definition 3. A sequence Q of events *covers* another sequence Q' of events if all the events in Q' appear in Q in the same order as they appear in Q'.

In the above definition, it is important to note that the events in Q' do not have to appear consecutively in Q. For example, a partial test sequence [*open*, *read*, *write*] covers three 2-way sequences: <*open*, *read*>, <*open*, *write*> and <*read*, *write*>.

Definition 4. A *t*-way target sequence Q is a *t*-way sequence that can be covered by at least one test sequence.

Note that not every *t*-way sequence is a target sequence. Consider the File API example. If a repetition constraint requires that no event can be repeated, then 2-way sequences <*open*, *open*> and <*close*, *open*> cannot be covered by any test sequence. Thus, these sequences are not 2-way target sequences under given constraints.

Definition 5. Let Π be the set of all the *t*-way target sequences. A *t*-way test sequence set Σ is a set of test sequences such that for $\forall \pi \in \Pi$, $\exists Q \in \Sigma$ such that Q covers π.

Considering the motivating example, a set of two test sequences {[*open*, *read*, *write*, *close*], [*open*, *write*, *read*, *close*]} covers all 2-way target sequences <*open*, close>, <*open*, *read*>, <*open*, *write*>, <*read*, *close*>, <*read*, *write*>, <*write*, *close*> and <*write*, *read*>.

Note that the above definitions are similar to some earlier work in [7], which is however based on LTS.

Comparing to the previous algorithm of *t*-way test sequence generation [31], we optimize validity and extensibility check by using DFA. The primary task is to translate a set of user-given constraints specified using our notation into a DFA.

5.4.2 Translation from Constraints to DFA

As shown in Fig. 5-7, the translation from constraints to DFA is as follows: Assume that there is a sequencing model consisting of events and sequencing constraints. First, for each constraint, we translate it into a DFA based on the semantics of Regular and Sequencing operators; Then, we get the intersection of automata from all constraints to be the final DFA;

Finally, to simplify this DFA, we remove its transitions which have meaningless labels (may brought in by Complement operations) and merge its states through Minimize operation on automaton.

```
Automaton translateModelToDFA(M)
Input: A sequencing model M = <E, S>, where E is a set of events and S is a set of sequencing constraints
Output: The DFA A translated from sequencing model
{
   let A be the automaton accepting any sequence of events in E    // "(∀e)*", abbr. "e*"
   for (each sequencing constraint C in set S)
      A = A.intersection(translateConToDFA(E, C))
   remove transitions having meaningless labels from A
   A.minimize()    // Minimize A by merging its states
   return A
}

Automaton translateConToDFA(E, C)
Input: (a) a set E of events; and (b) a sequencing constraint C
Output: The DFA A translated from constraint
{
   if (constraint C is a single event or event set e1)
      return the automaton accepting any single event in e1    // "e1"
   // When a constraint consists of an operator and two operands (or single operand)
   let op be the operator of constraint C
   let l be the left operand of constraint C    // Single operand is considered as left
   let r be the right operand of constraint C if exists
   let automaton B = translateConToDFA(E, l)
   let automaton C = translateConToDFA(E, r) if r exists
   // 1. Regular operators: NOT, CONCAT, AND, OR
   if (op is "!")    // "! B"
      return B.complement()
   if (op is ".")    // "B . C", abbr. "BC"
      return B.concatenate(C)
   if (op is "&&")    // "B && C"
      return B.intersection(C)
   if (op is "||")    // "B || C"
      return B.union(C)
```

```
  // 2. Sequencing operators: always operator, immediate operators
  let automaton A be the automaton accepting any sequence of events in E    // "e*"
  if (op is "_")    // "_ B" (a.k.a. "B") = "e*Be*"
     return A.concatenate(B).concatenate(A)
  if (op is "+-")    // "B +- C" = "!( e*B!(Ce*) )"
     return A.concatenate(B).concatenate(C.concatenate(A).complement()).complement()
  if (op is "-+")    // "B -+ C" = "!( !(e*B)Ce* )"
     return A.concatenate(B).complement().concatenate(C).concatenate(A).complement()
  if (op is "~")    // "B ~ C" = "!( e*BCe* )"
     return A.concatenate(B).concatenate(C).concatenate(A).complement()
}
```

Figure 5-7. Translation algorithm from constraints to DFA

In brief, when a constraint (supported by syntax) consists of one or two shorter constraints as operands via Regular operators, its corresponding automaton should be derived through regular operations (Complement, Concatenate, Intersection and Union) [23] on automata corresponding to operands. Likely, shorter constraints as operands via Sequencing operators should be handled in the similar way, except that the semantics of Sequencing operators are a bit more complicated. Note that the implementation code of regular operations on automata refers to a third-party library "dk.brics.automaton" [24].

5.4.3 Validity and Extensibility Check

Recall that there are three types of constraints, sequencing, repetition, and length constraints.

Our algorithm for checking whether a sequence is valid is shown in Fig. 5-8. Given a test sequence Q, we first check whether it satisfies all the repetition and length constraints by counting its number of events, then check whether it satisfies all the sequencing constraints by running it via DFA.

```
Boolean isValid(Q, A, C)
Input: (a) an input sequence Q; (b) the DFA A translated from a set of sequencing constraints; and (c) a set C of
other constraints such as the repetition and length constraints
Output: Whether Q is valid or not
{
  if (Q violates C)
    return false
  // If sequence Q is accepted by A, then return true; Otherwise, return false
  return A.run(Q)
}
```

Figure 5-8. Validity check algorithm

Our algorithm for checking whether a sequence is extendable is shown in Fig. 5-9. The algorithm employs a recursive Depth-First Search strategy, i.e., a sequence is extendable if there exists an event making the extended sequence with this event to be either valid or extendable, while itself satisfies repetition/length constraints. We adopt an optimization that the check does not try all events for extension, but only the outgoing events of the current state which the input sequence would arrives at inside DFA. By using this optimization, we reduce the time complexity of extensibility check from O(n^L) to O(k^L), n is the total number of events, k is the maximum number of outgoing events of each state in DFA, L is the maximum length of sequence. Since in most cases k is much smaller than n, the time complexity is greatly reduced.

```
Boolean isExtendable(Q, A, C)
Input: (a) an input sequence Q; (b) the DFA A translated from a set of sequencing constraints; and (c) a set C of
other constraints such as the repetition and length constraints
Output: Whether Q is extendable or not
{
  if (Q violates C)
    return false
  let ST be the state that Q arrives at inside A
  if (ST is Error state)
```

```
    return false
  // Traverse via outgoing edges of ST to find a possible event for extension
  for (each transition T of state ST) {
    let e be the event of transition T
    set Q' to be Q.e
    if (isValid(Q', A, C))
      return true
    else if (isExtendable(Q', A, C))
      return true
  }
  return false
}
```

Figure 5-9. Extensibility check algorithm

Note that the validity and extensibility check results of any sequence can be cached so that there is no redundant check.

5.4.4 Test Sequence Generation Algorithm

```
Pair<Set, Set> GenTestSeqs(M, C, t)
Input: (a) a sequencing model M = <E, S>, where E is a set of events and S is a set of sequencing constraints; (b) a set C of other constraints such as the repetition and length constraints; and (c) a test strength t
Output: (a) a t-way test sequence set Σ; and (b) its uncovered sequence set Π
{
    // Step 1: DFA translation
1.  let automaton A = translateModelToDFA(M)
    // Step 2: target sequence (candidate) generation
2.  let Π be {π = <e1, e2, …, et> | ei ∈ E (i=1...t) and π satisfies C}
    // Step 3: test sequence generation
3.  let Σ be an empty set
4.  while (Π is not empty) {
      // Step 3.1: starting phase
5.    create a starting test sequence Q such that (a) Q covers at least one target sequence in Π; and (b) isValid(Q, A, C)
          or isExtendable(Q, A, C)
6.    if (Q cannot be created)
7.      break   // Break the loop leaving some sequences in Π uncovered
8.    remove from Π the target sequences covered by Q
      // Step 3.2: extension phase
```

```
9.     while (isExtendable(Q, A, C)) {
10.      append an event e in E to Q such that (a) Q.e covers the most target sequences in Π; and (b) isValid(Q.e, A, C)
           or isExtendable(Q.e, A, C)
11.      Q = Q.e
12.      remove from Π the target sequences covered by Q
13.  }
14.  add Q into Σ
15.}
16.return <Σ, Π>
}
```

Figure 5-10. Algorithm GenTestSeqs

Fig. 5-10 shows our test sequence generation algorithm. The algorithm consists of three major steps. The step 1 is to translate the set of sequencing constraints into a DFA for further validity and extensibility check. The step 2 is to generate target sequence candidates which may be covered by generated test sequences. Note that not every sequence in Π is a target sequence, as some sequences in Π may not be covered by any test sequence due to the interactions between sequencing and repetition/length constraints. As discussed later, the step 3 guarantees that all the target sequences in Π will be covered. Thus, after the step 3, the remaining sequences in Π cannot be covered by any test sequence and are not target sequences. Note that we could check whether every sequence in Π is a target sequence and remove those that are not (by repeating step 3.1) prior to the step 3. This, however, is redundant.

In the step 3, we generate test sequences to cover all the target sequences. We first create a starting test sequence Q to cover at least one remaining target sequence. A starting test sequence must be valid or extendable. This is necessary to ensure termination. If such a test sequence cannot be created, all the target sequences in Π have already been covered, and the algorithm

terminates. Otherwise, we extend Q by appending events one by one. Each time we select an event *e* that makes Q.*e* covers the most target sequences in Π, while Q.*e* must be valid or extendable. When no event can be appended to Q, Q becomes valid and not-extendable, and we add Q into the resulting test set and create another starting test sequence. We continue to do so until we cannot find a starting sequence that covers at least one sequence in Π, or Π is empty.

We call the phase of creating a starting test sequence as a starting phase (step 3.1) in lines 5 - 8, and the phase of extending a test sequence to be valid and not-extendable as an extension phase (step 3.2) in lines 9 - 14. Note that test sequence is always extended to be not-extendable because that our current generation algorithm is a greedy algorithm considering a longer test sequence would cover more target sequences, which is not optimal as we know. Also, the sorting of events in E for extension affects the generated test sequences, i.e., with different orders to cover target sequences, the generation would produce different sets of test sequences. The problem how to figure out the best ordering of target sequences to be covered for optimal test sequence set is similar to the optimal test set problem in some previous research [33][34], however, is much more complicated so we plan to solve it in the future research.

5.4.5 Creation of Starting Sequence

In this part, we discuss the creation of a starting test sequence, i.e., a valid or extendable test sequence that covers at least one remaining *t*-way target sequence, within a reasonable time.

As indicated in Fig. 5-10 line 5, in order to ensure termination of test sequence generation, we create a starting test sequence that covers at least one target sequence in Π. The algorithm employs a Breadth-First Search strategy as shown in Fig. 5-11, which uses a queue to traverse all possible length *t* sequence candidates, then length *t*+1 and so on, until found a starting sequence.

```
Sequence createStartingTestSeq(E, A, C, t, Π)
Input: (a) a set E of events; (b) the DFA A translated from sequencing constraints; (c) a set C of other constraints such
as the repetition and length constraints; (d) a test strength t; and (e) a set Π of remaining target sequence candidates
Output: A starting test sequence Q
{
  // Initialize a queue U of all possible candidates for starting sequence
  let U be a queue consisting of all the event sequences of length t
  // Update the queue U by extension until a starting sequence is found
  while (U is not empty) {
    remove the first sequence Q from U
    if (Q covers at least one target sequence in Π) {
      if (isValid(Q, A, C) or isExtendable(Q, A, C))
        return Q
    }
    else if (isExtendable(Q, A, C)) {
      for (each event e in E)
        append e to Q and add it to the end of U
    }
  }
  return null
}
```

Figure 5-11. Algorithm to create a starting test sequence

Note that the worst case of this algorithm is unable to find any starting test sequence till the maximum length constrained by extensibility check. The time complexity of the worst case is O(*m*), *m* is the number of all valid or extendable sequences from length *t* to the maximum length L. (Recall that the validity and extensibility check results of each sequence can be cached for

reuse.) The worst case may not appear during test sequence generation if all target sequence candidates in Π can be covered by generated test sequences. If some candidates in Π are uncoverable, the worst case appears once as the termination of generation.

5.5 Case Study

In this section, we apply our test sequence generation framework to the IEEE 11073-20601 protocol (Optimized Exchange Protocol) [10]. As a core component in the standards family of IEEE 11073, this protocol defines a communication model that allows Personal Health Devices (PHDs) to exchange data with computing devices. In IEEE 11073, there are two types of devices, agent and manager devices. Agents are PHDs that are used to obtain measured health data from the user. Managers manage and process the data collected by agents.

In this case study, we express sequencing constraints which IEEE 11073 protocol imposes on its communication behavior using our notation and generate *t*-way test sequences that satisfy all these constraints. The generated set of test sequences can be used to perform conformance testing of an implementation of the protocol, e.g., Antidote [11].

5.5.1 Sequencing Constraints

We identify constraints from the manager's perspective. Note that constraints can be similarly identified from the agent's perspective. In particular, as participants of the same protocol, agent and manager exhibit to a large extent symmetrical behavior, in terms that a send event on one side corresponds to a receive event on the other side.

Comparing to the sequencing constraints in [31], we reconsider the relationships of events because we allow events to connect in the form of label "input/output" in Mealy Machine [27]. These labels will become the *actual events* used in constraints and test sequence generation.

From the perspective of manager, when it receives a request from agent, it must immediately handle this request and transmit response to agent, i.e., using "RxReq/TxRsp" as actual events can abbreviate this kind of relationship between such type of two events. Note that events TxReq and RxRsp may not be immediately connected due to waiting for agent, so that they cannot use "/" to abbreviate their relationship. Thus, there is a total of 10 actual events which can be divided into three groups, based on their functional areas.

- Actual events that try to establish association:

Three possible requests from agent and their immediate responses by manager:

RxAssocReq_unacceptable_configuration/TxAssocRsp_rejected

RxAssocReq_acceptable_and_known_configuration/TxAssocRsp_accepted

RxAssocReq_acceptable_and_unknown_configuration/TxAssocRsp_accepted_unknown_config

- Actual events that try to accept configuration:

One request form agent which supplies a config to manager:

RxConfigEventReportReq

Two requests triggered by manager software interface (abbr. REQs) that first check whether the config supplied by agent is unsupported or supported, and then immediately transmit the response TxRsp corresponding to the above config request RxReq:

REQAgentSuppliedUnsupportedConfig/TxConfigEventReportRsp_unsupported_config

REQAgentSuppliedSupportedConfig/TxConfigEventReportRsp_accepted_config

- Actual events that try to release association:

One request from agent and its immediate response by manager:

RxAssocRelReq/TxAssocRelRsp

Three actual events respectively represent an effective REQ with its immediately transmitted request TxReq, a redundant REQ without transmitting TxReq, response RxRsp which would be not necessarily immediately received from agent:

REQAssocRel/TxAssocRelReq, *REQAssocRel*, *RxAssocRelRsp*

In our study, we first identify simple constraints from each functional group separately and then put them together to construct nested constraints. Based on our latest notation of operators and Nesting Feature, the sequencing constraints are specified as shown in Fig. 5-12 (For readability, we symbolize actual events by chars as shown in TABLE 5-3). Note that the DFA from our constraints has been proved equivalent to the FSM of IEEE 11073 manager which we could manually construct based on its manager state table [28].

TABLE 5-3. SYMBOL TABLE MAPPING ACTUAL EVENTS TO CHARS

Sym.	**Actual event**
a	*RxAssocReq_unacceptable_configuration/TxAssocRsp_rejected*
b	*RxAssocReq_acceptable_and_known_configuration/TxAssocRsp_accepted*
c	*RxAssocReq_acceptable_and_unknown_configuration/TxAssocRsp_accepted_unknown_config*
d	*RxConfigEventReportReq*
e	*REQAgentSuppliedUnsupportedConfig/TxConfigEventReportRsp_unsupported_config*
f	*REQAgentSuppliedSupportedConfig/TxConfigEventReportRsp_accepted_config*
g	*RxAssocRelReq/TxAssocRelRsp*
h	*REQAssocRel/TxAssocRelReq*
i	*REQAssocRel*
j	*RxAssocRelRsp*

```
1.  {a, b, c} ...+ {d, e, f, g, h, i, j}    // implied by other constraints so can be reduced
2.  {b, c, d, e, f, h, i} +... {a, g, j}
3.  (_ {c, e} && {c, e} .~. {d, g, h}) -+ d
4.  d -+ {e, f}
5.  h ... j    // "h ... j" is abbreviated from "h +- (_ j && h .~. j) && (_ h && h .~. j) -+ j"
6.  (_ {b, c} && {b, c} .~. {g, h}) -+ h
7.  (_ h && h .~. j) -+ i
8.  ((_ {b, c} && {b, c} .~. {g, h}) || (_ h && h .~. j)) -+ g
9.  ((_ {b, c} && {b, c} .~. {g, h}) || (_ h && h .~. j)) ~ {a, b, c}
```

Figure 5-12. All 9 sequencing constraints of PHD manager model

Constraint 1. Any of three *RxAssocReq/TxAssocReq* events is the possible first event that **must happen before** all other events, since try to establish association is the first step of any execution.

Constraint 2. Any of three events *RxAssocRelRsp*, *RxAssocRelReq/TxAssocRelRsp*, *RxAssocReq_unacceptable_configuration/TxAssocRsp_rejected* is the possible last event that **must happen after** all other events. These events indicate that association has been released or rejected.

Constraint 3. *RxConfigEventReportReq*, which receives a new configuration from the agent, **can only happen after** the manager transmitted a response that the previous configuration is checked to be unknown or unsupported, **meanwhile** neither any new configuration has been received nor any Association Release Request has happened.

Constraint 4. Two REQs about configuration checking **can only happen immediately after** the manager received a configuration from agent.

Constraint 5. *REQAssocRel/TxAssocRelReq* **must pair with** *RxAssocRelRsp*, since if the manager triggered association release request, it must wait the agent for response to this request. Otherwise, it is unsecure for PHDs, as the manager may stop service too early without the acknowledge of agent.

Constraint 6. *REQAssocRel/TxAssocRelReq*, which releases association from manager side, **can only happen** when the association is accepted and not yet released.

Constraint 7. When the manager triggers *REQAssocRel* for the first time, it will immediately transmit an Association Release Request and wait for an Association Release Response, i.e., actual event *REQAssocRel/TxAssocRelReq* happens and waits for pairing *RxAssocRelRsp*. During

the waiting, if the manager triggers a redundant *REQAssocRel*, no more requests would be sent. In other words, actual event *REQAssocRel* **can only happen after** *REQAssocRel/TxAssocRelReq* triggered, **meanwhile** *RxAssocRelRsp* has not yet responded.

Constraint 8. *RxAssocRelReq/TxAssocRelRsp*, which releases association from agent side, **can only happen either** when the association is accepted and not yet released, **or** when the association is released by manager but not yet acknowledged by agent.

Constraint 9. Any of three *RxAssocReq/TxAssocReq* events, which establishes association from agent side, **never happen either** when the association is accepted and not yet released, **or** when the association is released by manager but not yet acknowledged by agent. Note that the nesting constraint in left operand is the same as Constraint 8.

Note that one benefit of our notation is it allows incremental specification. That is, we do not require all the constraints be specified up front. Instead, we can begin with several constraints, translate the set of these constraints into a DFA, and then check whether the DFA is as expected. If not, we can add more constraints. This can be repeated for multiple times until we capture all the constraints.

Also note that there lacks the measurement of use-cost to compare our notation with other existing notations such as FSM. The challenge is that it is hard to measure how much effort or knowledge is required when users adopt different notations for sequential testing on their systems.

We plan to solve the problem how to design a general measurement method for different notations specifying sequencing constraints in the future research.

5.5.2 Repetition and Length Constraint Settings

Recall that besides of sequencing constraints, we also introduce repetition and length constraints for generation to control the maximum length of test sequences.

The maximum repetition *r* of repetition constraint "Rep ≤ *r*" usually should be no less than one and no more than strength *t*. However, test sequence generation sometimes cannot achieve *t*-way coverage with Rep ≤ *t*, in which case *r* must be set as more than *t*. For example, in order to satisfy an uncommon sequencing constraint "(a.a.a) ...+ b", 2-way target sequence <a, b> requires subsequence [a, a] immediately before it to become sequence [..., a, a, a, ..., b, ...] which violates Rep ≤ 2.

The maximum length *max* of length constraint "Len ≤ *max*" must be no less than *t* (since a test sequence must cover at least one *t*-way target sequence) and no more than $r \times N$ (*N* is the number of all events in SUT).

5.5.3 Test Sequence Generation Results

The experimental environment is set up as the following: OS: Windows 10, CPU: Intel Core i5 1.6GHz, Memory: 8 GB, SDK: Java SE 1.7.

We use the 9 sequencing constraints in Fig. 5-12 with different repetition and length constraints (as shown in the first two columns of TABLE 5-4 and 5-5) to generate test sequences that achieve 2-way and 3-way sequence coverage.

TABLE 5-4. RESULTS OF 2-WAY TEST SEQUENCE GENERATION

Rep cons	Len cons	# of 2-way seqs	# of covered seqs	Gen Time (sec)	# of test seqs	test seq length			Sum of test lens
						min	avg	max	
≤ 2	≤ 2	100	3	0.003	3	2	2.0	2	6
≤ 2	≤ 3	100	16	0.005	8	2	2.9	3	23
≤ 2	≤ 4	100	43	0.021	17	4	4.0	4	68
≤ 2	≤ 5	100	69	0.059	23	4	5.0	5	114
≤ 2	≤ 6	100	88	0.052	27	6	6.0	6	162
≤ 2	≤ 7	100	96	0.075	24	6	6.9	7	165
≤ 2	**≤ 8**	**100**	**100**	**0.087**	**19**	**6**	**7.9**	**8**	**150**
≤ 2	**≤ 9**	**100**	**100**	**0.049**	**18**	**8**	**8.9**	**9**	**161**
≤ 2	**≤ 10**	**100**	**100**	**0.037**	**12**	**8**	**9.7**	**10**	**116**
≤ 2	**≤ 11**	**100**	**100**	**0.026**	**11**	**10**	**10.6**	**11**	**117**
≤ 2	**≤ 12**	**100**	**100**	**0.049**	**10**	**10**	**11.3**	**12**	**113**
≤ 2	**≤ 13**	**100**	**100**	**0.066**	**10**	**11**	**12.5**	**13**	**125**
≤ 2	**≤ 14**	**100**	**100**	**0.050**	**8**	**12**	**13.4**	**14**	**107**
≤ 2	**≤ 15**	**100**	**100**	**0.088**	**11**	**12**	**14.1**	**15**	**155**
≤ 2	**≤ 16**	**100**	**100**	**0.052**	**10**	**12**	**15.3**	**16**	**153**
≤ 2	**≤ 17**	**100**	**100**	**0.052**	**6**	**15**	**16.0**	**17**	**96**
≤ 2	**≤ 18**	**100**	**100**	**0.038**	**6**	**18**	**18.0**	**18**	**108**
≤ 2	**≤ 19**	**100**	**100**	**0.044**	**6**	**18**	**18.0**	**18**	**108**
≤ 2	**≤ 20**	**100**	**100**	**0.040**	**6**	**18**	**18.0**	**18**	**108**

TABLE 5-4 shows that, for 2-way sequence coverage (i.e., to cover all target sequences of any 2 of 10 events), when repetition constraint is set to be Rep≤ 2, the number of 2-way sequences is 10×10 = 100, and all can be covered by generation under proper length constraint.

Note that we mark result entries in bold which cover all *t*-way sequences, and underline entries which have generated a same set of test sequences whose lengths cannot be longer.

We convert marked entries into Fig. 5-13 which shows that, when the length constraint is being relaxed, the number of generated test sequences goes down from 19 to 6 (while their maximum length increases from 8 to 18 following length constraint). However, there is a rebound of number when maximum length is from 15 to 16, which indicates that longer test sequences may not always lead to smaller test sequence set. The trends of two clustered columns and one line finally go flat when the length constraint is relaxed enough (i.e., Len≤18, which is 9×2 determined by sequencing constraints with repetition constraint) that does not affect the generation anymore. According to these trends, we suggest choosing Len≤12, Len≤14 or Len≤17 as proper length constraint to generate a local optimal set of test sequences for 2-way coverage, under given sequencing constraints and repetition constraint.

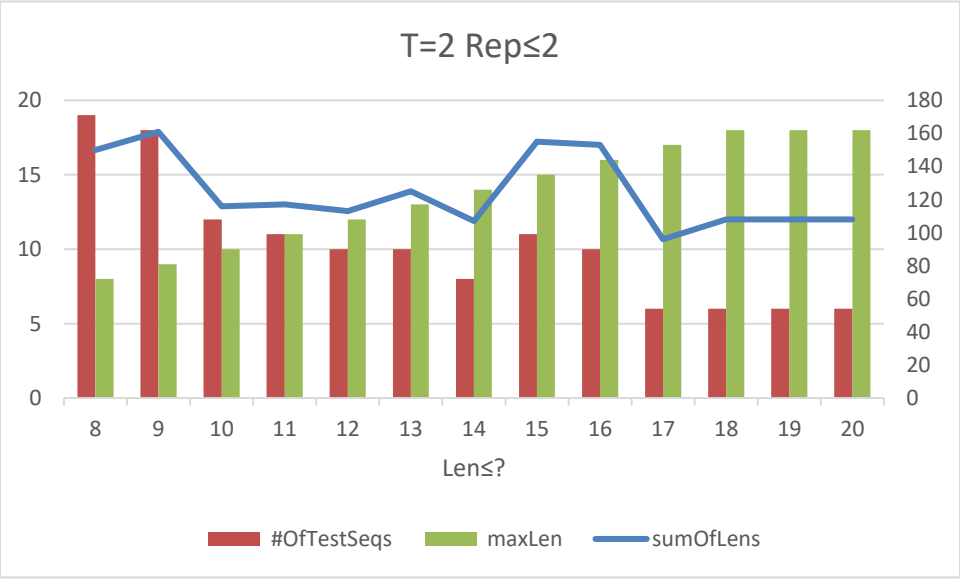

Figure 5-13. Trends of generated test sequences with 2-way coverage

TABLE 5-5. RESULTS OF 3-WAY TEST SEQUENCE GENERATION

Rep cons	Len cons	# of 3-way seqs	# of covered seqs	Gen Time (sec)	# of test seqs	test seq length			Sum of test lens
						min	avg	max	
≤ 2	≤ 10	990	946	7.106	120	9	10.0	10	1196
≤ 2	≤ 11	990	965	5.614	100	9	10.9	11	1088
≤ 2	**≤ 12**	**990**	**967**	**5.864**	**70**	**10**	**11.8**	**12**	**825**
≤ 2	**≤ 13**	**990**	**967**	**6.733**	**73**	**11**	**12.7**	**13**	**929**
≤ 2	**≤ 14**	**990**	**967**	**8.660**	**59**	**12**	**13.5**	**14**	**797**
≤ 2	**≤ 15**	**990**	**967**	**13.466**	**72**	**12**	**14.0**	**15**	**1011**
≤ 2	**≤ 16**	**990**	**967**	**17.028**	**63**	**12**	**14.9**	**16**	**940**
≤ 2	**≤ 17**	**990**	**967**	**20.154**	**54**	**12**	**15.8**	**17**	**851**
≤ 2	**≤ 18**	**990**	**967**	**21.206**	**48**	**12**	**16.6**	**18**	**799**
≤ 2	**≤ 19**	**990**	**967**	**21.603**	**48**	**12**	**16.6**	**18**	**799**
≤ 2	**≤ 20**	**990**	**967**	**21.653**	**48**	**12**	**16.6**	**18**	**799**
≤ 3	≤ 10	1000	986	7.777	139	9	10.0	10	1388
≤ 3	≤ 11	1000	996	5.428	96	10	11.0	11	1054
≤ 3	**≤ 12**	**1000**	**1000**	**5.163**	**76**	**11**	**11.9**	**12**	**906**
≤ 3	**≤ 13**	**1000**	**1000**	**4.210**	**64**	**12**	**13.0**	**13**	**831**
≤ 3	**≤ 14**	**1000**	**1000**	**3.833**	**58**	**12**	**13.8**	**14**	**798**
≤ 3	**≤ 15**	**1000**	**1000**	**3.341**	**52**	**13**	**14.8**	**15**	**768**
≤ 3	**≤ 16**	**1000**	**1000**	**3.431**	**45**	**14**	**15.6**	**16**	**700**
≤ 3	**≤ 17**	**1000**	**1000**	**2.938**	**40**	**15**	**16.3**	**17**	**653**
≤ 3	**≤ 18**	**1000**	**1000**	**3.096**	**48**	**16**	**17.7**	**18**	**848**
≤ 3	**≤ 19**	**1000**	**1000**	**2.124**	**33**	**17**	**18.5**	**19**	**612**
≤ 3	**≤ 20**	**1000**	**1000**	**1.947**	**34**	**18**	**19.3**	**20**	**656**
≤ 3	**≤ 21**	**1000**	**1000**	**1.788**	**41**	**19**	**20.7**	**21**	**849**
≤ 3	**≤ 22**	**1000**	**1000**	**2.139**	**38**	**20**	**21.6**	**22**	**819**
≤ 3	**≤ 23**	**1000**	**1000**	**2.156**	**38**	**18**	**22.2**	**23**	**845**
≤ 3	**≤ 24**	**1000**	**1000**	**1.966**	**52**	**18**	**23.6**	**24**	**1225**
≤ 3	**≤ 25**	**1000**	**1000**	**1.918**	**44**	**23**	**24.2**	**25**	**1066**
≤ 3	**≤ 26**	**1000**	**1000**	**2.167**	**44**	**24**	**25.0**	**26**	**1101**
≤ 3	**≤ 27**	**1000**	**1000**	**0.668**	**42**	**25**	**26.8**	**27**	**1124**
≤ 3	**≤ 28**	**1000**	**1000**	**0.735**	**42**	**25**	**26.8**	**27**	**1124**
≤ 3	**≤ 29**	**1000**	**1000**	**0.735**	**42**	**25**	**26.8**	**27**	**1124**
≤ 3	**≤ 30**	**1000**	**1000**	**0.717**	**42**	**25**	**26.8**	**27**	**1124**

TABLE 5-5 shows that, for 3-way sequence coverage (i.e., all target sequences of any 3 of 10 events), when repetition is constrained to be no more than 2, the number of 3-way sequences is 10×10×10 - 10×1×1 = 990 (since event cannot align with itself twice due to Rep≤2), and the number of sequences that can be covered by generated test sequences is no more than 967 (since there are 23 target sequences cannot happen due to sequencing constraints and repetition constraints, e.g., a 3-way target sequence requires one of its events happening before it to satisfy sequencing constraints but violates Rep≤2, like target sequence <a, b, b> requires another prior event b to become sequence [..., b, ..., a, ..., b, ..., b, ...] whose maximum repetition > 2).

Note that the generation times in entries whose 3-way sequences are not fully covered are much longer than those entries covering all 3-way sequences. The reason is that, in the former entries, there exist the worst case to create a starting test sequence as the termination of their generations. The worst case consumes much more time than general case since it tries all valid or extendable sequences from length *t* to maximum length. It is also the reason why the generation times in the former entries increases with the increment of maximum sequence length.

We convert the marked entries with Rep≤2 into Fig. 5-14 which shows that, when the length constraint is being relaxed, the number of generated test sequences goes down from 70 to 48 (while their maximum length increases from 12 to 18 following length constraint). However, there are two rebounds of number when maximum length is 13 or from 15 to 16. The trends of two clustered columns and one line finally go flat when the length constraint is relaxed enough

(i.e., Len≤18, which is 9×2) that does not affect the generation anymore. According to these trends, we suggest choosing Len≤12, Len≤14 or Len≤18 as proper length constraint to generate an optimal set of test sequences for NOT fully 3-way coverage (since only 967 of 990 3-way sequences are covered), under given sequencing constraints and repetition constraint.

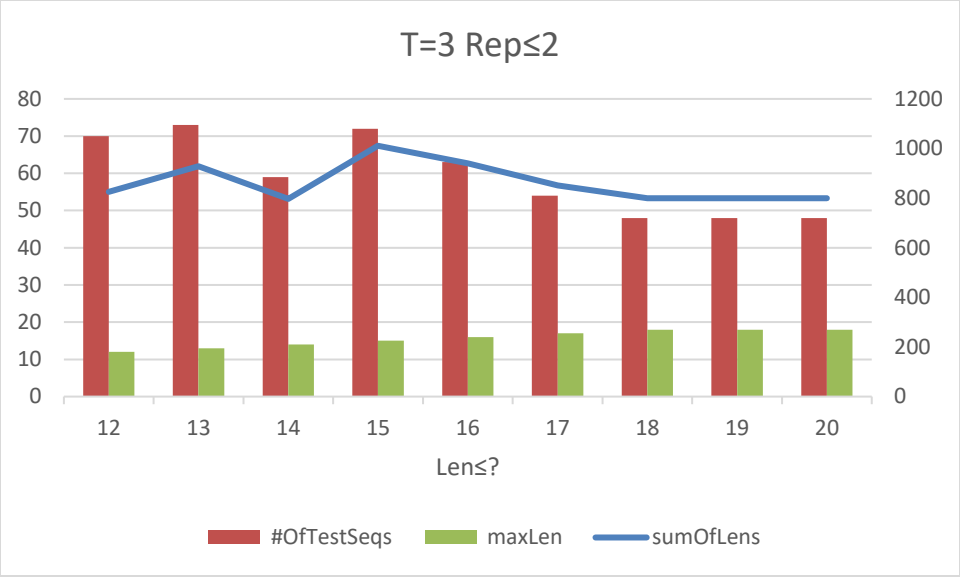

Figure 5-14. Trends of generated test sequences with NOT fully 3-way coverage

When repetition constraint is relaxed to be Rep≤3, the number of 3-way sequences is 10×10×10 = 1000, and all can be covered by generation under proper length constraint. We convert marked entries with Rep≤3 into Fig. 5-15 which shows that, when the length constraint is being relaxed, the number of generated test sequences first goes down from 76 to 33 (while their maximum length increases from 12 to 19 following length constraint), and then goes up from 34 to 42 (maximum length increases from 20 to 27). However, there are one rebound of number when maximum length is 18, and two surges when maximum length is 21 or 24. The trends of

two clustered columns and one line finally go flat when the length constraint is relaxed enough (i.e., Len≤27, which is 9×3) that does not affect the generation anymore. According to these trends, we suggest choosing Len≤17 or Len≤19 as proper length constraint to generate a local optimal set of test sequences for fully 3-way coverage, under given sequencing constraints and repetition constraint.

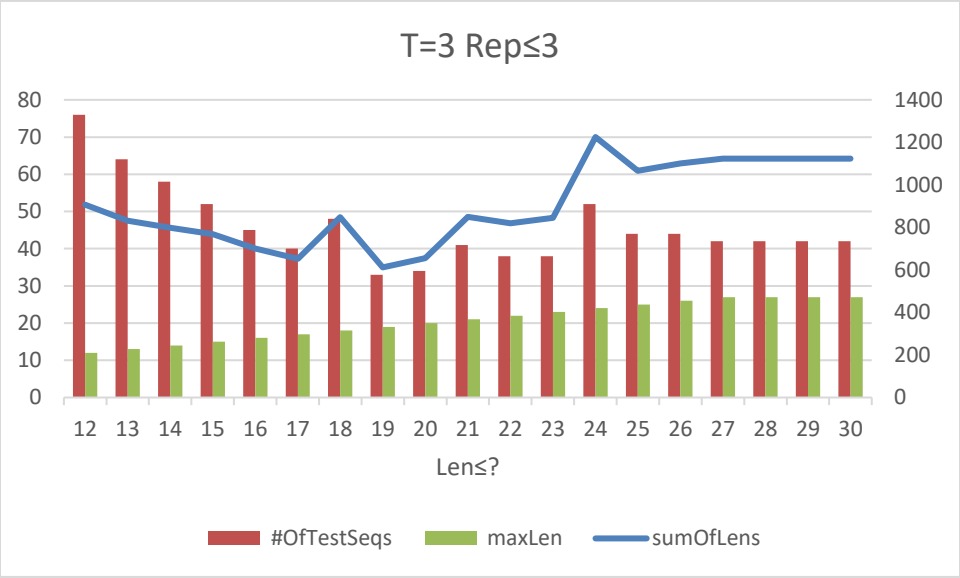

Figure 5-15. Trends of generated test sequences with fully 3-way coverage

Based on the above results, it is interesting to find that by setting a proper length constraint, the generated set of test sequences would be local optimal on the balance of the number of sequences (which determines the cost of test execution and evaluation) vs. sequence lengths (which may affect the complexity of debugging). Our approach to determine the proper length constraint for a SUT (which has already determined sequencing constraints, strength *t* and repetition constraint) can be divided into two steps: First relax the length constraint to achieve

t-way coverage; Then continue relaxing until a local optimal set of test sequences is generated whose number of sequences and sum of sequence lengths are local minimum.

5.6 Related Work

Combinatorial testing has been an active area of research [6]. However, most work has focused on t-way test data generation [12]. In this section, we focus our discussion on *t*-way sequence generation that supports constraints.

There exist many *t*-way test sequence generation approaches supporting constraints. However, some of them lack the capability to specify all possible constraints for real-life systems, while others require a low-level specification of constraints such as dependency graph or state transition diagram.

Kuhn et al. [4] presented an approach to generate *t*-way SCAs. Their approach requires each event to appear exactly once in a test sequence. Thus, the length of each test sequence is fixed which equals the number of events. It supports only one type of constraint on sequence as "*x*..*y*", which means that no test sequence should contain *x* and *y* in the given order, equivalent to our notation "*x* .~. *y*". There are certain types of constraints between two events that cannot be specified using this notation. Furthermore, their notation cannot specify constraints involving more than two events.

Farchi et al. [13] presented an approach to generate test sets that satisfy ordered and unordered interaction coverage. Their ordered restrictions can be considered as a type of

sequencing constraints. For example, an ordered restriction excluding a case "Read.comesBefore(Open)" is to prevent <Read, Open> from generation. This restriction is similar to the notation in [4], and thus has similar limitations as mentioned earlier.

Several approaches have been reported that use a graph model to represent system behavior from which *t*-way test sequences are generated. Wang et al. [14] presented a pairwise test sequence generation approach for web applications. Their approach is based on a graph model called navigation graph that captures the navigation structure of a web application. Rahman et al. [15] presented a test sequence generation approach using simulated annealing. Their approach is based on a state transition diagram that models the system behavior. Yu et al. [7][8] presented several algorithms that generate *t*-way test sequences from LTS models. Bombarda et al. [29] [30] presented automaton-based methods to generate combinatorial test sequence by using state machine. In these approaches, sequencing constraints are implicitly encoded in the graph model. Compared to our notation, the graph models used in these approaches are at a lower level of abstraction and require a lot of operational details that may not be readily available in practice.

Kruse et al. [16] suggested that temporal logic formulas, e.g., Linear Temporal Logic (LTL) [17], Computational Tree Logic (CTL) [18], and modal μ-calculus [19], can be used to express sequencing constraints. They used LTL for dependency rules (i.e., sequencing constraints) and CTL for generation rules (i.e., strength *t*, repetition and length constraints). Temporal logic formulas are powerful in terms of the different types of properties they could be used to express.

However, these notations have a complex semantic model, and have found limited use in practice. For example, both LTL and CTL have a state-based semantic model. In theory, any state-based property can be specified using events, and vice versa. However, the notion of state is more difficult to grasp than that of event. This is because unlike events, states are not directly represented in a test sequence. Thus, in order to specify sequencing constraints, events must be translated into states. This translation can be difficult due to the fact that states can be defined at different levels of abstraction and thus the mapping between states and events may not be a simple one-to-one relation.

Dwyer et al. [20] developed a system of property specification patterns to specify properties that are commonly encountered in practice. Our work is different in that we define a small set of operators, each of which captures a fundamental relationship between two events. Complex properties can be specified by nesting these operators. The work in [20] is complementary with ours in that similar patterns can also be identified to facilitate the use of our notation in practice.

5.7 Conclusion and Future Work

There seems to be significant interests on *t*-way sequence testing in both academia and industry. However, progress is still lacking. In this paper we present an approach to handling sequencing constraints, which we believe is a key technical challenge in *t*-way test sequence generation but has not been adequately addressed. Our approach consists of an event-oriented notation for expressing sequencing constraints and a greedy algorithm for generating test

sequences that achieve *t*-way coverage while ensuring that all the constraints are satisfied. Our notation is user-friendly to derive sequencing constraints from SUT, convenient to explain the nesting structure of complex constraint and can be translated to DFA based on automata operations. The DFA is used to perform validity and extensibility check of generation algorithm, which greatly reduce its time cost. We report a case study in which our notation and generation algorithm are applied to a real-life communication protocol that exhibits sequencing behavior. The experiment results show that our generation algorithm is practical from the perspectives of both generation time and generated set size.

In the future, we will continue our work on sequential test generation in the following major directions. First, we want to apply both our notation and test generation algorithm on a bunch of real-life SUTs to evaluate their effectiveness and correctness, by comparing experiment results and bug detection reports with other sequential testing tools using different notation and algorithm. Second, we want to define a general measurement method of use-cost for different notations specifying sequencing constraints. The measurement will be applied to measure how much time or effort is required when users adopt different notations for sequential testing on their systems. Finally, we want to solve the problem how to figure out the best order in which all target sequences would be covered one by one to form an optimal test sequence set, i.e., the extension of the optimal test set problem from combinatorial testing to sequential testing area.

5.8 References

[1] A. Canny. “Interactive system testing: beyond GUI testing.” In Proceedings of the 2018 ACM SIGCHI Symposium on Engineering Interactive Computing Systems, p. 18.

[2] R.C. Bryce, S. Sampath, and A.M. Memon. “Developing a single model and test prioritization strategies for event-driven software.” IEEE Transactions on Software Engineering 37, no. 1 (2011): 48-64.

[3] D.E. Simos, J. Bozic, B. Garn, M. Leithner, F. Duan, K. Kleine, Y. Lei, and F. Wotawa. “Testing TLS using planning-based combinatorial methods and execution framework.” Software Quality Journal (2018): 1-27.

[4] D.R. Kuhn, J.M. Higdon, J.F. Lawrence, R.N. Kacker, and Y. Lei. “Combinatorial methods for event sequence testing.” In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 601-609.

[5] F. J. Daniels, and K. C. Tai. “Measuring the effectiveness of method test sequences derived from sequencing constraints.” In Proceedings of the 1999 Technology of Object-Oriented Languages and Systems, pp. 74-83.

[6] C. Yilmaz, S. Fouche, M.B. Cohen, A. Porter, G. Demiroz, and U. Koc. “Moving forward with combinatorial interaction testing.” Computer 47, no. 2 (2014): 37-45.

[7] L. Yu, Y. Lei, R.N. 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 (ICECCS), pp. 220-229.

[8] L. Yu, Y. Lei, R.N. Kacker, D.R. Kuhn, R.D. Sriram, and K. Brady. “A general conformance testing framework for IEEE 11073 PHD's communication model.” In sixth International Conference on Pervasive Technologies Related to Assistive Environments (PETRA 2013), p. 12.

[9] L. Yu, Y. Lei, M. Nourozborazjany, R.N. Kacker, and D. R. Kuhn. “An efficient algorithm for constraint handling in combinatorial test generation.” In 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), pp. 242-251.

[10] J.H. Lim, C. Park, S.J. Park, and K.C. Lee, “ISO/IEEE 11073 PHD message generation toolkit to standardize healthcare device.” In 2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society. pp. 1161-1164.

[11] Antidote IEEE 11073-20601 stack library. [Online]. http://oss.signove.com/index.php/Antidote:_IEEE_11073-20601_stack

[12] K. Kleine, and D.E. Simos. “An efficient design and implementation of the In-Parameter-Order algorithm.” Mathematics in Computer Science 12, no. 1 (2018): 51-67.

[13] E. Farchi, I. Segall, R. Tzoref-Brill, and A. Zlotnick. “Combinatorial testing with order requirements.” In 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 118-127.

[14] W. Wang, S. Sampath, Y. Lei, and R.N. Kacker. “An interaction-based test sequence generation approach for testing web applications.” In Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium (HASE), pp. 209-218

[15] M. Rahman, R.R. Othman, R.B. Ahmad, and M.M. Rahman. “Event driven input sequence t-way test strategy using simulated annealing.” In 2014 5th International Conference on Intelligent Systems, Modelling and Simulation (ISMS), pp. 663-667.

[16] P.M. Kruse, and J. Wegener. “Test sequence generation from classification trees.” In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 539-548.

[17] A. Pnueli. “The temporal logic of programs.” In 1977 18th Annual Symposium on Foundations of Computer Science, pp. 46-57.

[18] E.M. Clarke, and E.A. Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logic.” In 1981 Workshop on Logic of Programs, pp. 52-71.

[19] D. Kozen. “Results on the propositional μ-calculus.” Theoretical computer science 27, no. 3 (1983): 333-354.

[20] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. “Patterns in property specifications for finite-state verification.” In Proceedings of 1999 21st International Conference on Software Engineering (ICSE), pp. 411-420.

[21] R. McNaughton, and H. Yamada. “Regular expressions and state graphs for automata.” IRE transactions on Electronic Computers 1 (1960): 39-47.

[22] L. Karttunen, J. P. Chanod, G. Grefenstette, and A. Schille. “Regular expressions for language engineering.” Natural Language Engineering 2, no. 4 (1996): 305-328.

[23] A. Fellah, H. Jürgensen, and S. Yu. “Constructions for alternating finite automata.” International journal of computer mathematics 35, no. 1-4 (1990): 117-132.

[24] dk.brics.automaton - finite-state automata and regular expressions for Java. [Online]. https://www.brics.dk/automaton/

[25] M. O. Rabin, and D. Scott. “Finite automata and their decision problems.” IBM journal of research and development 3, no. 2 (1959): 114-125.

[26] C. Chia-Hsiang, and R. Paige. “From regular expressions to DFA's using compressed NFA's.” Theoretical Computer Science 178, no. 1-2 (1997): 1-36.

[27] G. H. Mealy. “A method for synthesizing sequential circuits.” The Bell System Technical Journal 34, no. 5 (1955): 1045-1079.

[28] ISO/IEEE 11073-20601:2010 Health informatics - Personal health device communication - Part 20601: Application profile - Optimized exchange protocol. [Online]. https://www.iso.org/standard/54331.html

[29] A. Bombarda, S. Bonfanti, A. Gargantini, M. Radavelli, F. Duan, and Y. Lei. “Combining model refinement and test generation for conformance testing of the IEEE PHD protocol using abstract state machines.” In 2019 IFIP International Conference on Testing Software and Systems, pp. 67-85.

[30] A. Bombarda, and A. Gargantini. “An Automata-Based Generation Method for Combinatorial Sequence Testing of Finite State Machines.” In 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 157-166.

[31] F. Duan, Yu Lei, Raghu N. Kacker, and D. Richard Kuhn. “An approach to t-way test sequence generation with constraints.” In 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 241-250.

[32] D. R. Kuhn, Renee Bryce, Feng Duan, Laleh Sh Ghandehari, Yu Lei, and Raghu N. Kacker. “Combinatorial testing: Theory and practice.” Advances in Computers 99 (2015): 1-66.

[33] F. Duan, Y. Lei, L. Yu, R. N. Kacker, and D. R. Kuhn. “Improving IPOG's Vertical Growth Based on a Graph Coloring Scheme.” In 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1-8. IEEE, 2015.

[34] F. Duan, Y. Lei, L. Yu, R. N. Kacker, and D. R. Kuhn. “Optimizing IPOG's Vertical Growth with Constraints Based on Hypergraph Coloring.” In 10th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 181-188. IEEE, 2017.

CHAPTER 6.

CONCLUSION AND FUTURE WORK

In this dissertation, we present our research works on both combinatorial and sequential test generation.

In Chapter 2 to 3, we mainly present an approach improving the vertical growth phase of combinatorial test generation algorithm IPOG that reduces the size of its generated test set, starting from no constraint to supporting constraints. The improvement is based on the idea of greedy coloring approach on "Minimum Vertex Coloring" problem which vertical growth phase can be converted into, while supporting constraints requires the graph coloring problem be extended to hypergraph coloring since some constraint would be converted into hyperedges. The fundamental concept of coloring approach is that, in the graph/hypergraph model, vertices are either missing tuples waiting to be colored or existing tests already colored in distinct colors at the initial state; edges/hyperedges are conflicts among vertices that cannot be put in a same test. After coloring, a group of vertices in same color can be transformed to exactly a valid test. The experimental results indicate that this approach generates optimal test sets for many real-life systems with constraints.

In Chapter 4 to 5, we mainly present an event-oriented notation to specify sequencing constraints and an algorithm of *t*-way test sequence generation supporting this notation whose generation time cost is greatly reduced from start to end. Our notation is user-friendly to derive sequencing constraints from System Under Test (SUT), convenient to explain the nesting structure

of complex constraint and can be translated to Deterministic Finite Automaton (DFA) based on operations on automata. The DFA is adopted in sequence validity and extensibility check of generation algorithm, which greatly reduce its time cost. We report a case study in which our notation and generation algorithm are applied to a real-life communication protocol. The experiment results show that our generation algorithm is practical from the perspectives of both generation time and generated set size.

Our future work on sequential test generation would be in the following major directions. First, we want to apply our approach on a bunch of real-life SUTs to evaluate its effectiveness and correctness, by comparing with other sequential testing tools which consist of different notation and generation algorithm. Second, we want to develop a method to perform consistency check on constraints specified by the user. This consistency check can reject contradictory constraints prior to test generation and can also provide feedback to the user in terms of how to make corrections. Third, we want to investigate the formal properties of our notation for sequencing constraint specification, in terms of what kind of constraints our notation can or cannot express. In particular, we want to check the possible equivalence relation between our notation and other notations such as LTS and LTL, then design a general measurement method of use-cost for different notations. Finally, we want to solve the problem how to figure out the best order in which all target sequences are covered one by one to form an optimal test sequence set. This problem would be an expansion of the optimal test set problem in Chapter 2 and 3 from combinatorial testing to sequential testing area.