Efficient Algorithms for T-Way Test Sequence Generation

Linbin Yu[1], Yu Lei[1], Raghu N. Kacker[2], D. Richard Kuhn[2], James Lawrence[3]

[1]*Dept. of Computer Sci. and Eng.*
University of Texas at Arlington
Arlington, TX 76019, USA
{linbin.yu@mavs.uta.edu,
ylei@cse.uta.edu}

[2]*Information Technology Laboratory*
National Institute of Std. and Tech.
Gaithersburg, MD 20899, USA
{raghu.kacker, kuhn}@nist.gov

[3]*Dept. of Mathematical Sciences,*
George Mason University,
Fairfax, VA 22030, USA
lawrence@gmu.edu

***Abstract*—Combinatorial testing has been shown to be a very effective testing strategy. Most work on combinatorial testing focuses on t-way test data generation, where each test is an unordered set of parameter values. In this paper, we study the problem of t-way test sequence generation, where each test is an ordered sequence of events. Using a general labeled transition system as the system model, we formally define the notion of t-way sequence coverage, and introduce an efficient algorithm to compute all valid t-way target sequences, i.e., sequences of t events that must be covered by at least one test sequence. We then report several algorithms to generate a set of test sequences that achieves the proposed t-way sequence coverage. These algorithms are developed as the result of a systematic exploration of the possible approaches to t-way test sequence generation, and are compared both analytically and experimentally. The results show that while these algorithms have their own advantages and disadvantages, one of them is more scalable than others while exhibiting very good performance.**

Keywords: Combinatorial Testing; T-way Sequence Coverage; Test Sequence Generation;

I. INTRODUCTION

Combinatorial testing has been shown to be a very effective testing strategy [4-7]. Most work on combinatorial testing focuses on t-way test data generation, where each test is an (unordered) set of values for parameters. T-way combinatorial testing, or t-way testing, requires every combination of values for *any* t parameters be covered by at least one test. The rationale behind t-way testing is that many faults involve only a few parameters, thus testing all t-way combinations can effectively detect these faults. However, many programs exhibit sequence-related behaviors. For example, faults in graphical user interface (GUI) programs may only be triggered by a certain sequence of user actions [12]; faults in web applications may only be exposed when some pages are viewed in a certain order [9]; and faults in concurrent programs may not manifest unless some events are exercised in a particular order [3]. Testing efforts for these programs should not only focus on data inputs, but also sequences of actions or events.

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

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

To our best knowledge, our work is the first attempt to systematically study the problem of t-way test sequence generation using a general system model. The major contributions of this paper are as follows.

1) We define the t-way sequence coverage for a general system model that can be used to model different types of programs, such as GUI applications, web applications, and concurrent programs.
2) We propose a set of algorithms for t-way test sequence generation, including an efficient algorithm for generating valid t-way sequences that must be covered, and four algorithms for generating a small set of test sequences that achieve the t-way sequence coverage. These algorithms are implemented in a Java application which is freely available to the public [15].

3) We report an experimental evaluation of the proposed test generation algorithms. This evaluation provides important insights about the advantages and disadvantages of these proposed algorithms.

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

The remainder of this paper is organized as follows. Section II introduces preliminary concepts and motivation for t-way sequence testing. Section III presents an efficient algorithm for t-way target sequence generation. Several algorithms for test sequence generation are presented in Section IV. Experimental results are presented in Section V. In Section V we briefly overview related work. Finally, we conclude this paper and discuss future work in Section VI.

II. PRELIMINARIES

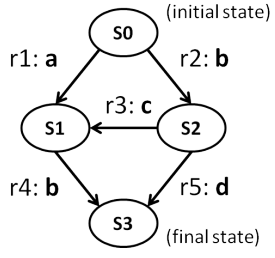
A. System Model

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

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

An LTS can be built from system requirements, high-/low-level designs, or implementations at a certain level of abstraction. The size of an LTS can be controlled by choosing an appropriate level of abstraction and by modeling system parts that are of interest, instead of the whole system. Initial states in an LTS represent states where an execution of system could start, and final states represent states where a system could safely stop.

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



(a)

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

(b)

Figure 1. (a) an exampleAn LTS graph (b) an exercised-after matrix

An LTS can be represented by a directed graph, in which each vertex represents a state, and each directed edge represents a transition between two states and is labeled with an event. An example graph is shown in Figure 1(a). We refer to such a graph as an LTS graph.

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

Intuitively, a transition sequence represents a path in an LTS graph. We use these two terms interchangeably. Given a transition sequence $P = r_1 \bullet r_2 \bullet ... \bullet r_n$, we denote the corresponding event sequence $event(P) = event(r_1) \bullet event(r_2) \bullet ... \bullet event(r_n)$. We use $event(P)$ to represent P if there is no ambiguity, i.e., only one test sequence can be represented by $event(P)$. We distinguish the notion of *transition sequence* from the notion of *sequence of transitions*. The former requires transitions to be consecutive in a sequence, whereas the latter does not require so. In the rest of this paper, we use $r_1 \bullet r_2 \bullet ... \bullet r_n$ to indicate a transition sequence, and use $< r_1, r_2, \ldots, r_n >$ to indicate a sequence of transitions.

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

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

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

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

Assume that there are n transitions in a labeled transition system. We can build a $n \times n$ matrix E, where $E(i,j)=1$ if $r_i \rightarrow r_j$, and $E(i,j)=0$ otherwise. This exercised-after matrix can be constructed in $O(n^3)$ time, using Warshall's algorithm [25]. This matrix will be used in t-way target sequence generation, as shown in Section III. Figure1 (b) shows the exercised-after matrix for the LTS graph in Figure 1(a).

B. T-way Sequence Coverage

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

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

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

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

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

A test sequence is a transition sequence that starts from an initial state and ends with a final state in the LTS graph. Intuitively, a test sequence is a (complete) transition sequence that can be exercised by a test execution.

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

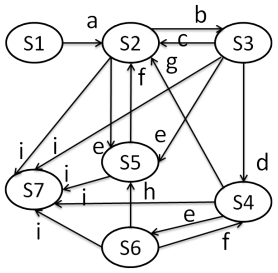
Intuitively, a target sequence Q is covered by a test sequence P if all the events in Q appear in $event(P)$ in order. For example, in Figure 1(a), a test sequence $r_2 \bullet r_3 \bullet r_4$ covers three 2-way target sequences: $<b, c>$, $<c, b>$, and $<b, b>$.

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

The t-way sequence coverage requires that every t-way target sequence be covered by at least one test sequence. Consider the example in Figure 1(a). Three test sequences, $r_1 \bullet r_4$, $r_2 \bullet r_5$ and $r_2 \bullet r_3 \bullet r_4$ covers all 2-way target sequences $<b, b>$, $<b, c>$, $<b, d>$, $<c, b>$ and $<a, b>$.

The notion of t-way sequence coverage is similar to *all-transition-k-tuples* coverage for web applications [11] and *length-n-event-sequence* coverage for GUI applications [10]. All these coverage criteria require sequences of a certain number of events be covered. However, they differ in that t-way sequence coverage does not require events in a target sequence to be covered consecutively by a test sequence whereas the other two require so. This difference has a significant implication on test sequence generation, which is illustrated below.

Figure 2 shows a labeled transition system that represents the life cycle of Java threads [16]. (This system is later used as a subject system in our experiments in Section V.A.)



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

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

Figure 2. Real Example: JavaThread

Assume that a fault can be exposed only if a sequence of three events, "start", "IO completes", "notify" (or event sequence $<a, h, g>$), is exercised. A shortest path (transition sequence) that covers $<a, h, g>$is $a \bullet b \bullet d \bullet e \bullet h \bullet f \bullet b \bullet d \bullet g$, which is of length 9. If *length-n-event-sequence* coverage is used, *all* paths of length 9 must be covered in order to cover this faulty sequence $<a, h, g>$. This can significantly increase the number of test sequences. In contrast, 3-way sequence testing may generate a transition sequence of length 9 for this 3-way target sequence, but it does not require all paths of length 9 be covered. This significantly reduces the number of test sequences while still detecting this fault.

III. TARGET SEQUENCE GENERATION

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

The main idea of our approach is to first generate all 2-way target sequences, and then extend 2-way target sequences to generate all 3-way target sequences, and continue to do so until we generate all the t-way target sequences. Given an LTS M, we first build the exercised-after matrix for all the transitions of M. Next we find the set of all possible transition pairs $<r, r'>$ that r' can be exercised after r. We refer to this set as the 2-way transition sequence set. For each 2-way transition sequence $<r, r'>$, and for each transition r'' that can be exercised after r', we generate a 3-way transition sequence $<r, r', r''>$. These 3-way transition sequences constitute the 3-way transition sequence set. We repeat this process until we build the t-way transition sequence set. At this point, we convert each t-way transition sequence to a t-way target sequence. Figure 3 shows the detailed algorithm that implements this approach.

Complexity analysis: Assume there are n transitions, and the test strength is t. There are at most n^t t-way transition sequences. As we discussed earlier, the time complexity for line 2 is $O(n^3)$, The time complexity for line 5 to 13 is

dominated by the last iteration, which is $O(n^t)$. Line 14 to 16 takes $O(tn^t)$ time. The time complexity of the entire algorithm is $O(n^3+tn^t)$, which is $O(n^3)$ when t<3, and $O(tn^t)$ otherwise. The space complexity is $O(n^2+tn^t)$ since we have to store the exercised-after matrix and all t-way transition sequences.

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

Figure 3. Algorithm for t-way sequences generation

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

IV. TEST SEQUENCE GENERATION

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

In this section, we present four algorithms for test sequence generation. These algorithms are developed from different perspectives. We compare them analytically in Section IV.E and experimentally in Section V.

A. A Target-Oriented Algorithm

Recall that in algorithm *GenTargetSeqs*, each t-way target sequence is derived from a sequence of *t* transitions (lines 14 and 15 in Figure 3). The main idea of this test generation algorithm is that, for each target sequence *Q*, we find a shortest test sequence to cover *Q* by extending the sequence of transitions from which *Q* is derived. Since a test sequence typically covers many target sequences, a greedy algorithm is then used to select a small subset of these test sequences that also covers all the target sequences.

The key challenge in this algorithm is how to generate a shortest test sequence to cover each target sequence. Let *Q* be a target sequence and *C* be the sequence of transitions from which *Q* is derived. We first extend *C* to a transition sequence *P* by inserting a transition sequence between every two adjacent transitions that are not consecutive. For example, assume $C = <r_1, r_2, ..., r_t>$. If r_{i+1} cannot be exercised immediately after r_i, we insert a shortest transition sequence between *dest*(r_i) and *src*(r_{i+1}). This path always exists (otherwise *Q* cannot be a t-way target sequence). Next, we make *P* a test sequence by inserting a shortest path from an initial state to the beginning state of *P* and a shortest path from the ending state of *P* to a final state, if necessary. The test sequence *P*, as constructed so, is a shortest test sequence that covers *Q*. Figure 4 shows the details of this algorithm.

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

Figure 4. A target-oriented algorithm for test sequences generation

Note that in line 2, all shortest paths between two nodes in a graph can be effectively calculated using Floyd-Warshall algorithm [21]. During target sequence generation, we keep all the sequence of transitions from which the target sequence is derived. These sequences are used in line 5. A greedy algorithm is used in line 17 to select a subset of test sequences that can cover all the target sequences. This algorithm is similar to a classic greedy set cover algorithm [21]. It works iteratively, i.e., at each iteration we choose a test sequence that covers the most number of uncovered target sequences, and remove covered ones, until all target sequences are covered.

Complexity analysis: Assume there are *n* transitions and $|\Sigma|$ target sequence in an LTS. Assume the test strength is *t*,

and m test sequences are generated. Finding all pair-wise shortest paths takes $O(n^3)$ time [21]. In order to build a shortest test sequence for each target sequence, we have to append at most $(t+1)$ shortest paths, which takes $O(t)$ time. Thus the total time from line 3 to 16 is $O(t|\Sigma|)$. The greedy algorithm in line 17 takes $O(m|\Sigma|^2)$ time. Therefore the time complexity for the entire algorithm is $O(n^3+t|\Sigma|+m|\Sigma|^2) = O(n^3+ m|\Sigma|^2)$. The space complexity is $O(n^2+|\Sigma|)$.

B. A Brute Force Algorithm

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

The first step of this algorithm uses a strategy similar to the breadth-first search (BFS) to generate all test sequences of length up to h. The difference is that in BFS, a node is only explored once, while in our strategy, a node is explored multiple times as it may lead to different test sequences. After all test sequences of length up to h are found, we apply the same greedy algorithm as mentioned in Section IV.A (Figure 4, line 17) to select a subset of test sequences to cover as many target sequences as possible. If there are any target sequences that remain uncovered, we use algorithm *GenTestSeqsFromTargets* to cover them. Figure 5 shows the detailed algorithm, named *GenTestSeqsBF*.

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

Figure 5. A brute force algorithm for test sequences generation

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

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

C. An Increamental Extension Algortihm

This algorithm is motivated by the observation that, a longer test sequence can cover more t-way targets. Theoretically, a test sequence of length l can cover as many as $C(l, t)$ t-way target sequences. Therefore longer sequences are preferred. However, the test sequences generated by algorithms *GenTestSeqsFromTargets* and *GenTestSeqsBF* are either selected from a set of shortest test sequences, or limited by the maximal length h. Therefore these sequences tend to be relatively short. This algorithm is designed to generate longer test sequences by extending a test sequence in an incremental manner. Once a long test sequence is generated, we remove covered targets and start generating another long test sequence. We keep doing so until no more targets can be covered. Similarly, the remaining targets are then covered by algorithm *GenTestSeqsFromTarget*.

The incremental extension is performed as follows. Given a transition sequence $P = r_1 \bullet r_2 \bullet ... \bullet r_n$, we generate the set Q of all *maximal* transition sequences that start from the ending state of this sequence, i.e., $dest(r_n)$, and that are of length up to h. A maximal transition sequence of length up to h is a transition sequence that is not a prefix of any other transition sequence of length up to h. A transition sequence P' is selected from Q such that sequence $P \bullet P'$ covers the most number of target sequences. Then we set $P = P \bullet P'$, *i.e.,* appending P' to P. An example of incremental extension is illustrated in Figure 6. In the previous extension (step *k-1*), a transition sequence that ends with transition a being the last transition was found. Now we explore all maximal transition sequences staring from $dest(a)$, and of length up to 3. Assume that a transition sequence $b \bullet c \bullet d$ is selected as it covers the most number of target sequences. This process is then repeated and the next extension will start from $dest(d)$. This extension is terminated when it reaches a final transition (successfully generated a long test sequence), or no targets can be covered (terminated).

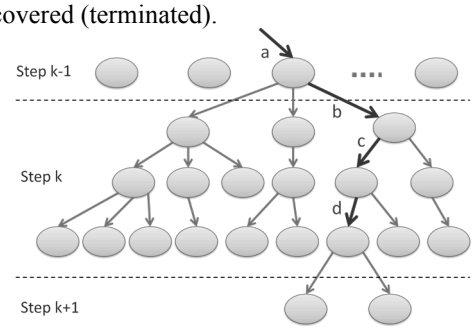


Figure 6. Illustration of test sequence extension.

Figure 7 shows the detailed *GenTestSeqsInc* algorithm. The parameter $h>t$ indicates the search depth in each extension. Note that the extension process may get stuck in a long cycle, and make no progress. In this case, the extension

process is terminated and the *GenTestSeqsFromTargets* algorithm is used to cover any remaining targets.

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

Figure 7. An increamental extension algorithm

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

D. An SCC-Base Algortihm

One of the biggest challenges in test sequence generation is how to handle transition cycles, as they could cause transition sequences to be extended infinitely. To address this problem, we either limit the maximal length of each test sequences (algorithm *GenTestSeqsBF*), or we terminate extension process when we are stuck in a cycle (algorithm *GenTestSeqsInc*). We use a different strategy to treat cycles in this SCC-based algorithm, which has three major steps. In the first step, we build an acyclic LTS M' from the original LTS M. In the second step, we find all necessary test sequences in the acyclic LTS M', referred as *abstract paths* as they need to be mapped back to the original LTS later. In the last step, we extend all the abstract paths to test sequences for the original LTS M, such that all target sequences are covered. We will explain each step with more details.

1) Build Acyclic LTS

In graph theory, a strongly connected component (SCC) is defined as a set of nodes in which any two nodes can reach each other. An SCC detection algorithm can be found in [21]. We can collapse each SCC in a LTS graph to a special node, called an SCC node. Doing this converts the original LTS to an acyclic LTS. An example is shown in Figure 8.

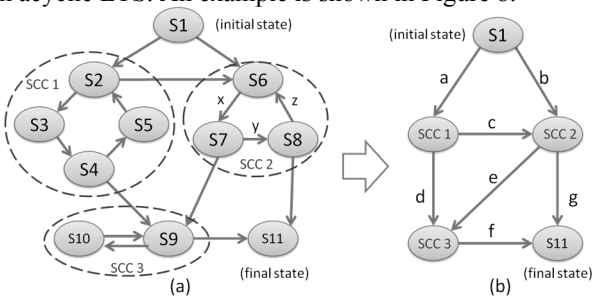


Figure 8. Example of SCC in an LTS graph

We generalize some important properties for SCCs which will be used later: (1) There exists a path from any node to another node in an SCC; (2) Any sequence of t events in an SCC is a t-way target sequence; (3) All the t-way target sequences in an SCC can be covered by a single transition sequence. Property (1) is derived from the definition of SCC. For property (2), given a sequence of t events, there exists a path that traverses t proper transitions in the given order and thus covers the given sequence of t events. Therefore any sequence of t events is a t-way target sequence. Property (3) is the key point for the last step of this SCC-based algorithm. We denote a single transition sequence that covers all t-way target sequences in an SCC as a *t-touring* path, which can be generated using the following approach. First we build a 1-touring path P_1, i.e., a path that traverses all transitions *at least* once in an SCC. Note that it's different with an *Eulerian path* which requires every edge to be visited *exactly* once. Then starting from the last transition of P_1, we build another 1-touring path P_2 and append P_2 to P_1. By doing this, we make a 2-touring path. This process is repeated until we have a t-touring path. It can be verified that a t-touring path contains a sequence of t 1-touring path, and therefore it covers all sequence of t events in the given SCC. For example, in Figure 8(a), a 3-touring path for SCC2 is $x \bullet y \bullet z \bullet x \bullet y \bullet z \bullet x \bullet y \bullet z$ which covers all 3-way target sequences consisting of events x, y and z. A detailed algorithm for t-touring path generation is described in Figure 9.

Algorithm: **GenTouringPath**
Input: an SCC M, strength t, state p, state q
Output: a t-touring path P from state p to state q
1. find a shortest path between any two states in M
2. let P be an empty transition sequence
3. let state $s = p$
4. for (i = 1 to t) {
5. let T be all transitions in M
6. while (T is not empty){
7. let r be a transition in T

```
8.        add into P a shortest path H from s to src(r)
9.        remove from T any transitions traversed by H
10.       let s be the last state of P
11.   }
12. }
13. add into P a shortest path H from s to r
14. return P
```

Figure 9. An algortihm for t-touring path generation

2) Find Abstract Paths

The abstract paths are actually test sequences of the acyclic LTS *M'*, which can be generated using any test generation algorithm such as algorithm *GenTestSeqsBF* and *GenTestSeqsInc*. The main challenge is that, we have to find all necessary abstract paths in *M'* such that, all target sequence of *M* can be covered by the test sequences derived from these abstract paths. To derive a test sequence from an abstract path, we simply insert a t-touring path after each transition ends with an SCC node. Therefore in abstract path generation, we have to maintain the set of target sequences of *M* which can be potentially covered.

3) Generate Test Sequences

As we discussed in step 2, we extend each abstract path by inserting a t-touring path after each transition ends with an SCC node, to extend it to a test sequence for M. Let $S = s_1 \bullet s_2 \bullet \ldots \bullet s_n$ be an abstract path of *M'*. If $dest(s_i)$ is an SCC node, we insert a t-touring path *P* from $dest(s_i)$ to $src(s_{i+1})$. Similar insertions are made for all the transitions that end with an SCC node. In Figure 8 (b), there are five abstract paths: *a•d•f*, *a•c•e•f*, *a•c•g*, *b•e•f* and *b•g*. Assume the test strength is 3. For path *b•g*, we insert a 3-touring path *x•y•z•x•y•z•x•y•z•x•y* after *b*, making a test sequence of the original LTS *b•x•y•z•x•y•z•x•y•z•x•y•z•x•y•g*. Similar insertions are made for other abstract paths. The whole SCC-based algorithm *GenTestSeqsSCC* is presented in Figure 10.

```
Algorithm: GenTestSeqsSCC
Input: an LTS M, a set Σ of target sequences, test strength t
Output: a t-way test sequence set Π for M
1. build an acyclic graph M' by finding and collapsing all SCCs
      in M
2. generate a set Ω consisting of abstract paths in M'
3. for (each abstract path P = c1 • c2 • ... • cn in Ω){
4.      for( i = 1 to n-1) {
5.          if( dest(ci) is an SCC node){
6.              use algorithm GenTouringPath to generate a t-
                    touring path T from dest(ci) to src(ci+1),
7.              insert T into P after transition ci
8.          }
9.      }
10.     if (the length of P is no less than t)
11.         add P to Π
12. }
13. if Π cannot cover all targets in Σ, generate a test sequences
    set Π' to cover remaining targets using algorithm
    GenTestSeqsFromTargets
14.   Π = Π ∪ Π'
15. return Π
```

Figure 10. An SCC-base algortihm for test sequences generation

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

E. Comparison of Test Generation Algorithms

We provide a brief analytic comparison of the four test sequence generation algorithms in Table 1.

TABLE I. COMPARISON OF TEST GENERATION ALGORITHMS

Algorithm	Time cost	Space Cost	Length of single test	# of generate tests
GenTestSeqsFromTargets	Low	Low	Short	Many
GenTestSeqsBF	High	High	Depends	Few
GenTestSeqsInc	High	Low	Long	Few
GenTestSeqsSCC	Low	Low	Very Long	Very Few

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

V. EXPERIMENTS

We have built a tool that implements the proposed algorithms. This tool is written in Java, and is freely available to the public [15]. To evaluate the performance of the proposed algorithms, we conducted experiments on both real and synthesized systems on a laptop with Corei5-2410M 2.30GHz CPU and 4GB memory, running Windows 7 (64-bit) and Java 6 SE (32-bit) with the default heap size.

The performances of the test generation algorithms are compared based on three metrics: the total length of generated test sequences, the number of generated test sequences, and the execution time taken to generate the test sequences.

A. Case Study: The Java Threads System

In this study we used the labeled transition system shown in Figure 2. Recall that this system describes the lifecycle of a Java thread, and it contains 7 states, 9 events and 16 transitions. The test strength is set to 3 in the experiments. There are 448 unique 3-way target sequences. Table II shows the results of applying the four test generation algorithms to this system. The columns in Table II are self-explanatory. For algorithm *GenTestSeqsInc*, we set *h*=5. For algorithm

GenTestSeqsBF, we use h=16, 18, 20 in three different runs, since 16 is the critical length, i.e., the smallest h such that all target sequences will be covered by test sequences of length up to h.

TABLE II. RESULT OF THE JAVATHREADS SYSTEM

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

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

B. Synthesied Systems

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

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

TABLE III. CHARACTERISTICS OF SYNTHESIZED SYSTEMS

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

Table IV shows the number of t-way target sequences generated by algorithm *GenTargetSeqs*, where t = 2, 3, 4, 5. It's obvious that the number of t-way target sequences increases very fast as test strength t and number of events increase.

TABLE IV. RESULTS OF T-WAY TARGET SEQUENCE GENERATION FOR SYSTEHSIZED SYSTEMS

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

TABLE V. RESULTS OF TEST SEQUENCE GENERATION FOR SYNTHESIZED SYSTEMS (3-WAY, 100% COVERAGE)

System	*GenTestSeqsFromTargets*				*GenTestSeqsBF*				*GenTestSeqsInc*				*GenTestSeqsSCC*			
	total length	# of test seqs	avg. length	time (s)	total length	# of test seqs	avg. length	time (s)	total length	# of test seqs	avg. length	time (s)	total length	# of test seqs	avg. length	time (s)
SYS-10	184	16	11.5	0.01	55	3	18.3	4.1	123	11	11.1	0.01	55	3	18.3	0.01
SYS-15	273	26	10.5	0.01	80	4	20	9.6	297	22	13.5	0.03	84	4	21.0	0.08
SYS-20	5346	547	9.8	0.09	2083	164	12.7	84.5	477	16	29.8	1.3	74	1	74.0	0.2
SYS-25	5049	446	11.3	0.1	1534	104	14.8	693.9	1066	62	19.4	2.6	188	4	47.0	0.2
SYS-30	8833	890	9.9	0.2	-	-	-	-	1381	55	25.1	32.5	132	2	66.0	0.4
SYS-40	35432	4163	8.5	3.0	-	-	-	-	7832	729	10.7	412.8	132	2	66.0	0.2
SYS-50	60047	5192	11.2	5.4	-	-	-	-	44058	1680	26.2	1080.7	439	4	109.8	2.4
SYS-60	113391	11097	10.0	30.7	-	-	-	-	-	-	-	-	403	3	134.3	6.2
SYS-80	211975	18829	11.3	110.8	-	-	-	-	-	-	-	-	354	2	177.0	10.2
SYS-100	500022	47166	10.6	526.0	-	-	-	-	-	-	-	-	557	3	185.7	14.6

C. Results and discussions

Table V shows the test generation results of synthesized systems using four proposed algorithms. The test strength *t* is set to 3, and the columns in Table V are self-explanatory. The coverage is verified by an independent procedure. Note that "-" indicates the process of test sequence generation took more than one hour to complete or ran out of memory with the default heap size. For algorithm *GenTestSeqsBF*, we set *h* to the critical length and increase it until timeout or out-of-memory. The *h* finally used in the first four systems is 30, 25, 15 and 13, respectively. For algorithm *GenTestSeqsInc*, we set *h*=5 such that every extension takes a reasonable time. For algorithm *GenTestSeqsSCC* we use *GenTestSeqsInc* to generate abstract paths, and the search depth is set to 10 in our experiments.

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

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

D. Threats to Validity

The external threat to validity is mainly due to the fact that the subject systems used in our experiments may not be representative of true practice. We used a real-life system and a number of randomly generated systems to reduce this threat. The internal threat to validity is mainly due to the fact that mistakes that could be made when conducting the experiments. We tried to automate the process as much as possible to reduce chances for human errors. We have also tried to cross-check the correctness of our results whenever there was a reasonable doubt.

VI. Related Work

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

A. Combinatorial Test Data Generation

Different strategies for combinatorial test data generation have been proposed in recent years. These strategies can be roughly classified into two groups, computational methods and algebraic methods [17, 22]. Computational methods usually involve an explicit enumeration of all possible combinations, and employ a greedy or heuristic search strategy to select tests. Examples of these methods include AETG [18], IPOG [19], and methods based on simulated annealing and hill climbing [22]. In contrast, algebraic methods build a t-way test set based on some pre-defined formulas, without enumerating any combinations. Examples of algebraic methods include orthogonal arrays [13, 24] and doubling-construction [14]. A survey on combinatorial test generation methods can be found in [2].

Although computational methods are more expensive than algebraic methods, they can be applied to general system configurations, whereas algebraic methods cannot. Nonetheless, test data generation methods cannot be directly applied to test sequence generation, due to some fundamental differences between the two problems discussed in Section I.

B. Test Sequence Generation

Several efforts have been reported that try to apply the idea of combinatorial testing to test systems that exhibit sequence-related behaviors. Wang *et al.* [8, 9] introduced a combinatorial approach to test dynamic web applications, which mainly investigates the fault detection effectiveness of a notion called pairwise interaction coverage on web applications. This is different from our work in that our focus is on efficient algorithms for test sequence generation.

X. Yuan *et al.* introduced a covering array method for GUI test case generation in [23]. This method first generates a covering array for abstract GUI events, and then generates executable sequences from this covering array. A more in-depth study on GUI testing is presented in [24], in which they incorporated the context of event. Similar to the work in [8, 9], their work mainly investigates the fault detection effectiveness of different coverage criteria. The test generation methods used in their work are based on covering arrays. All test sequences are extended from some fixed-length smoke tests. Our algorithms do not impose similar restrictions.

C. Test Sequence Coverage Criteria

We focus on coverage criteria that require "sequence of elements" to be covered. In other words, we do not consider coverage criteria such as all-nodes and all-branches [20].

Pairwise interaction coverage is used in some literatures such as Wang *et al.* [8, 9], which requires all possible pair of web page interactions to be covered by at least once. This is the special case of t-way sequence testing when t=2. Lucca and Penta [11] applied several coverage criteria for web

application testing from [1]. One of their criteria is *all-transition-k-tuples*, which requires all possible sequence of *k* consecutive transitions to be covered. Similarly, a coverage criterion, called *length-n-event-sequence* coverage, was proposed for GUI testing [10]. These coverage criteria require a sequence of elements to be covered consecutively. This is different from our t-way sequence coverage criterion, which only requires a sequence of elements to be covered in the same order, not necessarily consecutively.

The coverage criterion that is most closely related to ours is called *t*-coverage* coverage [24]. This coverage is proposed for GUI interaction testing and requires all permutation of *t* events are executed consecutively at least once and inconsecutively at least once. In t-way sequence coverage proposed in this paper, a sequence of elements is considered to be covered as long as it is covered once, consecutively or inconsecutively.

VII. Conclusion And future work

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

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

References

[1] R. Binder, Testing Object-Oriented Systems. AddisonWesley, 2000.

[2] M. Grindal, J. Offutt, and S. F. Andler, Combination Testing Strategies: A Survey. Journal of Software Testing, Verification and Reliability, 15, (2), pp. 97-133, 2005.

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

[4] D. R. Wallace, D. R. Kuhn, Failure modes in medical device software: An analysis of 15 years of recall data. International Journal of Reliability, Quality and Safety Engineering 2001; 8(4):351–371.

[5] D. R. Kuhn, M. J. Reilly, An investigation of the applicability of design of experiments to software testing. Proceedings of 27th NASA/IEEE Software Engineering Workshop, Greenbelt, Maryland, 2002; 91–95.

[6] Y.W. Tung, W. S. Aldiwan, Automating test case generation for the new generation mission software system. Proceedings of IEEE Aerospace Conference, Big Sky, Montana, 2000; 431–437.

[7] D. R. Kuhn, D. R. Wallace, A. J. Gallo, Jr., Software fault interactions and implications for software testing. Software Engineering, IEEE Transactions on, Volume: 30 Issue:6, 2004.

[8] W. Wang, S. Sampath, Y. Lei, R. Kacker, An Interaction-Based Test Sequence Generation Approach for Testing Web Applications. High Assurance Systems Engineering Symposium. Nanjing, 3-5 Dec. 2008.

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

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

[11] G. D. Lucca, M. D. Penta, Considering Browser Interaction in Web Application Testing. In 5th International Workshop on Web Site Evolution, pp. 74-81, 2003.

[12] P. Brooks, B. Robinson, and A. M. Memon, An initial characterization of industrial graphical user interface systems. Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation. Washington, DC, USA, 2009.

[13] A. W. Williams, Determination of test configurations for pair-wise interaction coverage. Proceedings of 13th International Conference on the Testing of Communicating Systems, Ottawa, Canada, 2000.

[14] M. A. Chateauneuf, C. J. Colbourn, D. L. Kreher, Covering arrays of strength 3. Designs, Codes, and Cryptography 1999;16:235–242.

[15] http://barbie.uta.edu/~lyu/seq

[16] http://docs.oracle.com/javase/

[17] A. Hartman, L. Raskin, Problems and algorithms for covering arrays. Discrete Mathematics 2004; 284(1–3):149–156.

[18] D. M. Cohen, S. R. Dalal, M. L. Fredman, G. C. Patton, The AETG system: An approach to testing based on combinatorial design. IEEE Transactions on Software Engineering 1997; 23(7):437–444.

[19] 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, 2007.

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

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

[22] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, W. B. Mugridge, Constructing test suites for interaction testing. Proceedings of 25th IEEE International Conference on Software Engineering, Portland, Oregon, 2003.

[23] X. Yuan, M. B. Cohen, and A. M. Memon, Covering Array Sampling of Input Event Sequences for Automated GUI Testing. In 22nd IEEE/ACM international conference on Automated Software Engineering, pp. 405-408, 2007.

[24] X. Yuan, M. B. Cohen. and A. M. Memon, GUI Interaction Testing: Incorporating Event Context. IEEE Transactions on Software Engineering, vol. NN, no. N, 2011.

[25] S. Warshall, A theorem on boolean matrices. Journal of the ACM, Volume 9, Number 1, pp. 11-12, 1962.

[26] R. Mandl, Orthogonal Latin squares: An application of experiment design to compiler testing. Communications of the ACM 1985; 28(10):1054–1058.