SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2007; **17**:207–225
Published online 7 June 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr.369

A combinatorial testing strategy for concurrent programs

Yu Lei[1,*,†], Richard H. Carver[2], Raghu Kacker[3] and David Kung[1]

[1]*Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76019-0015, U.S.A.*
[2]*Department of Computer Science, George Mason University, Fairfax, VA 22030, U.S.A.*
[3]*Information Technology Library, National Institute of Standards and Technology, Gaithersburg, MD 20899-8910, U.S.A.*

SUMMARY

One approach to testing concurrent programs is called reachability testing, which derives test sequences automatically and on-the-fly, without constructing a static model. Existing reachability testing algorithms are exhaustive in that they are intended to exercise all possible synchronization sequences of a concurrent program with a given input. In this paper, we present a new testing strategy, called t-way reachability testing, that adopts the dynamic framework of reachability testing but selectively exercises a subset of synchronization sequences. The selection of the synchronization sequences is based on a combinatorial testing strategy called t-way testing. We present an algorithm that implements t-way reachability testing, and report the results of several case studies that were conducted to evaluate its effectiveness. The results indicate that t-way reachability testing can substantially reduce the number of synchronization sequences exercised during reachability testing while still effectively detecting faults. Copyright © 2007 John Wiley & Sons, Ltd.

Received 8 September 2006; Revised 9 February 2007; Accepted 8 April 2007

KEY WORDS: combinatorial testing; concurrency testing; t-way testing; software testing

1. INTRODUCTION

Concurrent programming is becoming more popular in modern software development. A concurrent program contains two or more threads that execute in parallel and work together to perform

*Correspondence to: Yu Lei, Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76019-0015, U.S.A.
†E-mail: ylei@cse.uta.edu

Contract/grant sponsor: National Institute of Standards and Technology (NIST); contract/grant number: 60NANB6D6192

some task. Using multiple threads, a.k.a. multithreading, can increase computational efficiency. For instance, while one thread is waiting for user input, another thread can perform computational tasks in the background. Moreover, many problem domains are, by nature, concurrent and can be solved more naturally by creating multiple threads. As an example, a web server typically creates separate threads to service incoming client requests. Finally, as CPU manufacturers switch from uniprocessor to multi-core architectures, more applications will be structured as concurrent programs in order to benefit from the throughput advances brought about by these new architectures.

While concurrent programs offer some advantages they also exhibit non-deterministic behaviour, making them notoriously difficult to test. Multiple executions of a concurrent program with the *same* input may exercise *different* sequences of synchronization events (or SYN-sequences) and may produce *different* results. (The types of synchronization events that can appear in a SYN-sequence include send and receive events on communication channels, P and V events on semaphores, etc.) This non-deterministic behaviour is caused by factors such as variations in thread scheduling and in message latencies. One way to deal with non-deterministic behaviour during testing is to execute the program with the same input many times and hope that faults will be exposed by at least one of the executions. This type of uncontrolled testing, called *non-deterministic testing*, is easy to carry out, but it can be very inefficient [1]. It is possible that some SYN-sequences of the program will be exercised many times while others will never be exercised at all. *Deterministic testing* is an alternative approach in which program executions are controlled so that user-generated SYN-sequences can be tested [1]. One commonly suggested approach to generating test sequences is to select SYN-sequences from a static model of the program (or its design).[‡] However, static models are often inaccurate and/or may be too large to build for many applications [2].

Reachability testing is an approach that combines non-deterministic and deterministic testing [3–5]. A novel aspect of reachability testing is that it adopts a dynamic framework in which test sequences are generated automatically, and on-the-fly, without constructing a static model. During reachability testing, each test run is monitored and the SYN-sequence that is exercised is recorded in an execution trace. At the end of a test run, race analysis is performed to identify the race conditions and to derive the *race variants* of the SYN-sequence recorded in the trace. A race variant is derived by changing the race outcome of one or more race conditions, and it represents the beginning part of a SYN-sequence that could have happened but did not happen, due to the way race conditions were arbitrarily resolved during execution. These race variants are used to exercise new complete SYN-sequences, which are then analysed to derive new race variants, and so on. This process is repeated until no new SYN-sequence can be exercised.

Existing reachability testing strategies are exhaustive, i.e. they are intended to exercise every (partially ordered) SYN-sequence of a concurrent program with a given input exactly once. Exhaustive testing has important theoretical implications, but is often impractical due to resource constraints. In this paper, we describe a new reachability testing strategy. This strategy adopts the dynamic framework of reachability testing, but it exercises SYN-sequences selectively, not exhaustively. The selection of SYN-sequences is based on a combinatorial testing strategy known as t-way testing. Exhaustive reachability testing derives race variants to cover all possible combinations of the

[‡]A static model of a concurrent program is a model (or abstraction) of the program that is constructed by using static analysis, i.e. without actually executing the program.

DOI: 10.1002/stvr

race outcome changes that can be made in a SYN-sequence. In contrast, the new testing strategy, which we will refer to as t-way reachability testing, derives race variants to cover all possible t-way combinations of the race outcome changes, i.e. those involving changes to the outcomes of any t race conditions, where t is usually a small number, in a SYN-sequence. The hypothesis behind this new strategy is that not every race outcome contributes to every fault, and many faults can be exposed by interactions between a small number of race outcomes. We will present an algorithm that implements t-way reachability testing and report the results of several case studies that were conducted to evaluate its effectiveness. The results indicate that t-way reachability testing can substantially reduce the number of SYN-sequences exercised during reachability testing while still effectively detecting faults. For example, for a solution to the distributed dining philosophers (DDP) problem with three processes, exhaustive reachability testing exercises more than 5 million sequences, while 1-way reachability testing exercises only 5199 sequences on average. Despite this dramatic decrease in the number of sequences exercised, 1-way reachability testing is still able to detect all 303 faults seeded into faulty versions of DDP.

The remainder of the paper is organized as follows. Section 2 briefly reviews existing work on testing concurrent programs and on t-way testing. Section 3 illustrates the reachability testing process. Section 4 describes the t-way reachability testing strategy, and presents an algorithm that implements the strategy. Section 5 reports empirical results obtained by applying t-way reachability testing to several concurrent programs. Section 6 provides concluding remarks and describes our plan for future work.

2. RELATED WORK

In this section, we briefly review existing work on testing concurrent programs and on t-way testing.

2.1. Concurrency testing

Since t-way reachability testing is an implementation-based approach, we focus here on approaches that do not require formal specifications. As mentioned earlier, existing approaches to testing concurrent programs can be classified as either *non-deterministic* or *deterministic*, depending on whether test executions are controlled or not. Existing approaches to deterministic testing can be further split into two categories: *coverage-based testing*, which exercises a set of test sequences to satisfy a selected coverage criterion, and *state space exploration*, which systematically explores the state space of a program (or of a model of the program).

Non-deterministic testing: The main challenge for this approach is how to increase the chance of exercising different SYN-sequences during multiple, uncontrolled executions. The more the SYN-sequences are exercised, the better the chance will be of finding faults. One common approach to non-deterministic testing is to modify the subject program by inserting a set of 'noise makers', e.g. random delay statements [6], or calls to certain scheduling functions such as the Java functions *Thread.sleep*() and *Thread.yield*() [7,8]. It is hoped that the scheduling changes produced by these noise makers will cause different SYN-sequences to be exercised during repeated executions.

Non-deterministic testing is relatively easy to perform, but due to the lack of precise control, some SYN-sequences may be exercised many times, which is inefficient, while others may never

DOI: 10.1002/stvr

be exercised at all, which can cause faults to go undetected. In contrast, t-way reachability testing selectively exercises a set of SYN-sequences based on a well-defined strategy, namely t-way testing, and is guaranteed to exercise a different SYN-sequence each time the program is executed.

Coverage-based testing: This approach typically selects test sequences from a static model to satisfy a coverage criterion and then forces these sequences to be exercised during deterministic test runs. Two types of static models have been used for selecting SYN-sequences. The first type is called an extended control-flow graph (ECFG), which consists of a group of control-flow graphs (one control-flow graph for each thread) that are inter-connected by their static synchronization structure [9,10]. The second type is called a concurrency graph in which each node is a concurrency state, i.e. a state that displays the next synchronization-related activity to occur in each of the threads, and each edge is a transition between two concurrency states [11]. There are two problems with using a static model for selecting test sequences. First, test sequences selected from a static model may be infeasible, i.e. they cannot actually be exercised by any program execution. Second, it is difficult to capture certain dynamic behaviours, e.g. dynamic creation of threads and/or data structures, in a statically constructed model.

T-way reachability testing can be considered to be a coverage-based technique, but it differs from existing coverage-based testing techniques in that it derives test sequences automatically, and on-the-fly, without constructing a static model. Thus, t-way reachability testing does not have the above-mentioned problems with exercising infeasible paths and modelling dynamic behaviours. Also note that the selection of SYN-sequences in t-way reachability testing is based on a combinatorial strategy, namely t-way testing, which is different from the selection strategy used by existing coverage-based testing techniques.

State space exploration: Traditionally, this approach has been used to verify finite-state systems at the specification level. Several recent efforts have tried to apply state space exploration at the implementation level. Tools like Bandera translate a Java program into a Promela model and then use existing model checkers such as SPIN to explore the state space of the Promela model [12]. Program-to-model translation, however, can be difficult due to the semantic gap between programming languages and modelling languages. To overcome this problem, tools like Java PathFinder [13] and VeriSoft [14] directly explore the state space of actual programs, i.e. without any program-to-model translation. To alleviate the state explosion problem, partial order reduction techniques are employed to avoid exercising redundant paths that correspond to the same partial order.

State space exploration is based on an interleaving-based concurrency model in which every path explored is a totally-ordered sequence of events. To create a totally-ordered sequence, concurrent events are put into an arbitrary order, i.e. they are interleaved. This interleaving may create a blowup in the number of paths that have to be analyzed. In contrast, as shown in Section 3, t-way reachability testing is based on a true concurrency model that deals with partial orders directly. That is, concurrent events are recognized and left unordered. As a result, t-way reachability testing never generates two sequences that only differ in the order of concurrent events. Moreover, existing state exploration techniques are intended to be exhaustive. For large state spaces, exhaustive state exploration is impractical, and non-exhaustive state exploration may be ineffective if state exploration is forced to stop at arbitrary points. In contrast, t-way reachability testing is non-exhaustive and uses a well-defined strategy to selectively exercise test sequences.

Timeliness testing: Several studies have been reported on testing the timeliness of real-time applications [15–18]. While concurrency testing and timeliness testing both deal with temporal

DOI: 10.1002/stvr

properties, there is a fundamental difference between them. Concurrency testing does not have an explicit notion of time. Instead, it expresses temporal properties in terms of a relative ordering of events, e.g. an event e must happen before an event f. Thus, concurrency testing focuses on exploring different orderings of events. In contrast, timeliness testing deals with temporal properties that explicitly reference clock time, e.g. an event e must happen before a specific point in time or within a given time interval. Since the order of events in an execution directly affects the timing of these events, timeliness testing often requires timing constraints to be checked on different orderings of events. In this respect, our work is complementary to existing work on testing timeliness.

2.2. T-way testing

Combinatorial testing selects input values for individual parameters and combines these values to create tests. One strategy for combinatorial testing, called t-way testing, requires every possible combination of values of any t parameters to be included in some test case, where t is typically less than the total number of parameters. The key observation behind t-way testing is that not every parameter contributes to every fault, and many faults can be exposed by considering interactions among a small number of parameters.

Most approaches to t-way testing involve explicitly enumerating the combinations that need to be covered. Cohen *et al*. proposed a test generation strategy, called AETG (Automatic Efficient Test Generator), which constructs a test set by repeatedly adding one complete test at a time until all the combinations are covered [19]. A complete test is a test that assigns a value to every parameter. A greedy algorithm is used to construct the tests such that each test covers as many uncovered combinations as possible. Several variants of this strategy have been reported in the literature [20,21]. These variants share the same framework as AETG but use slightly different heuristics in the greedy construction of each test [22]. An alternative strategy, called IPO (or In-Parameter-Order), was described in [23,24]. The IPO strategy builds a t-way test set for the first t parameters, extends the test set to cover the first $t + 1$ parameters, and continues to extend the test set until it builds a t-way test set for all the parameters. Unlike in the AETG strategy, the tests added in IPO are not complete until the last parameter is covered. Covering one parameter at a time allows the IPO strategy to achieve a lower order of complexity than AETG [23]. Most recently, AI-based search techniques such as hill climbing and simulated annealing have been applied to t-way testing [25]. Unlike AETG and IPO, which build a test set from scratch, AI-based search techniques start from a randomly generated test set and then apply a series of transformations to the test set until a test set is reached that covers all the combinations. AI-based search techniques can produce smaller test sets than AETG and IPO, but they typically take significantly longer to complete [25]. Note that our work on t-way reachability testing adapts the IPO strategy to fit the framework of reachability testing.

The application of t-way testing to general software applications has been studied extensively in recent years. Burr and Young [26], showed that 2-way (or pairwise) testing can achieve a higher block and decision coverage than traditional methods for a commercial email system. Dalal *et al*. [27] reported a case study in which t-way testing was applied to a telephone system and showed that several faults can only be detected under certain combinations of input parameters. Kuhn *et al*. [28] studied the actual faults in several open source software projects, and found that all the known faults are caused by interactions among six or fewer parameters. Grindal *et al*. [29] reported an

DOI: 10.1002/stvr

empirical comparison of several t-way testing techniques, in terms of their test set sizes and fault detection effectiveness.

Note that a special case of t-way testing is 1-way testing, which requires that every value of every parameter be covered by at least one test case. While the general t-way testing algorithms mentioned earlier can be applied to 1-way testing, simpler algorithms could be developed for 1-way testing. Ammann and Offutt [30] proposed a 1-way coverage criterion called *base-choice-coverage* and described a procedure for constructing test sets that satisfy this criterion. The *base-choice-coverage* criterion was shown to be very effective in detecting faults [29].

To the best of our knowledge, no work has been reported on applying t-way testing to concurrent programs.

3. OVERVIEW OF REACHABILITY TESTING

In this section, we illustrate the reachability testing process and summarize the main results of our previous work on reachability testing [5]. Figure 1(a) shows a concurrent program CP that consists of four threads. The threads interact by sending messages to, and receiving messages from, ports. A port is a communication object that can be accessed by multiple senders and one receiver [31]. In program CP, port p_i has thread T_i as its receiver. Each send operation specifies a port as its destination, and each receive operation specifies a port as its source. Figure 1(b) shows a scenario in which reachability testing is applied to CP. Before we explain the details of this scenario, we define several important reachability testing concepts, namely *SYN-sequence*, *race*, and *race variant*.

A concurrent execution can be characterized by the source code, the input, and the SYN-sequence exercised by the execution. For a message-passing program, a *SYN-sequence* is defined to be a (partially ordered) sequence of send and receive events, as well as the synchronization relation between the events. A send event s is synchronized with a receive event r if the message sent by event s is received by event r. In this case, event s is said to be the send partner of event r, and event r the receive partner of event s. Figure 1(b) shows four SYN-sequences that could be exercised by

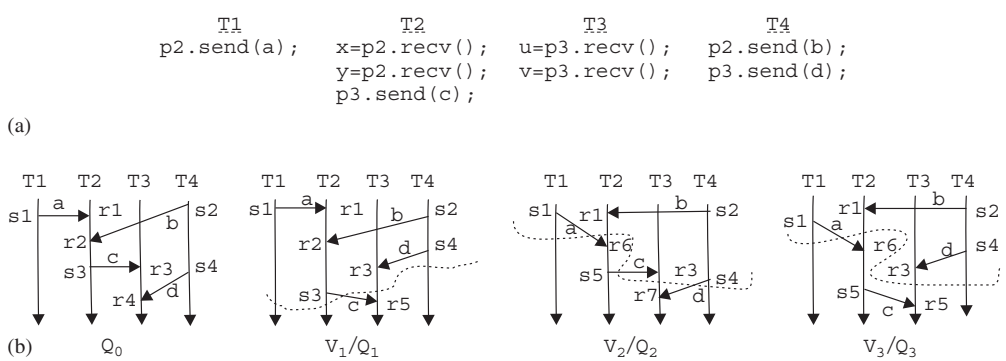

Figure 1. An example reachability testing process: (a) a message-passing program CP and (b) SYN-sequences and race variants exercised by reachability testing of CP.

DOI: 10.1002/stvr

program CP. Each SYN-sequence is depicted in a space–time diagram in which a vertical line represents a thread, and a single-headed arrow represents a message passed asynchronously from a send event to a receive event.

Let r be a receive event in a SYN-sequence Q. Let s be the send partner of r. Another send event s' is said to have a *message race* or simply a *race* with s if the message sent by s' could be received by r in a different execution. The race set of r, denoted as *race_set*(r), consists of all the send events that have a race with s. For example, in Q_0, race_set(r_1) = $\{s_2\}$, since r_1 could receive the message sent by s_2 instead of s_1, due to possible variations in message delay.

A race variant of a SYN-sequence Q is a (partially ordered) prefix of Q in which the outcomes of one or more races have been changed. The changes to the race outcomes need to be made in a way that satisfies the following constraints. Suppose that a receive event r receives a message from a send event s in a SYN-sequence Q, and is changed to receive a message from a send event s' in a race variant V of Q. Then, (1) there must exist a race between s and s' in Q; and (2) an event in Q must be removed from V if and only if its existence might be affected by this change. Note that the second constraint is needed to ensure that the sequence of events in a race variant can be exercised by at least one program execution. A formal definition of a race variant is given in [5]. A detailed discussion on how to generate race variants is presented in Section 4.2.

Now we are ready to explain the reachability testing process for program CP in Figure 1(a). It begins by executing CP non-deterministically, which we assume exercises SYN-sequence Q_0. Next, the races in Q_0 are identified and the outcomes of one or more races are changed to derive the 'race variants' of Q_0, namely V_1, V_2, and V_3. Variant V_1 is derived by changing the send partner of r_3 from s_3 to s_4, variant V_2 is derived by changing the send partner of r_1 from s_1 to s_2, and variant V_3 is derived by changing both the send partner of r_3 from s_3 to s_4 and the send partner of r_1 from s_1 to s_2. Each variant is used to conduct a prefix-based test run, which forces the events and synchronizations in the variant to be replayed and then allows the test run to proceed non-deterministically (i.e. without controlling which SYN-sequence is exercised). Prefix-based testing with V_1, V_2, and V_3 exercises sequences Q_1, Q_2, and Q_3, respectively. (In Figure 1(b) the sequence Q_i and the variant V_i used to exercise it are shown in the same space–time diagram. The events in the variant are the events above the dashed line.) No new variants will be derived from Q_1, Q_2, and Q_3, so the reachability testing process stops. Note that Q_0, Q_1, Q_2, and Q_3 are all the (partially ordered) SYN-sequences that the example program can possibly exercise.

In [5], we reported a general model that allows reachability testing to be applied to commonly used synchronization constructs, including semaphores, locks, monitors, and message passing. In addition, we reported a reachability testing algorithm that systematically exercises every SYN-sequence of a concurrent program with a given input exactly once, without saving a history of the SYN-sequences that have already been exercised. For the purpose of evaluation, we built a prototype reachability testing tool called RichTest and compared the performance of RichTest and VeriSoft. VeriSoft (from Bell Labs) [14] allows programs to be exhaustively tested, similar to RichTest, but VeriSoft is based on an interleaving-based concurrency model, while RichTest uses a true-concurrency model. In terms of the number of SYN-sequences that are required to be exercised during exhaustive testing, RichTest performed significantly better than VeriSoft for the programs we studied [5].

DOI: 10.1002/stvr

4. T-WAY REACHABILITY TESTING

4.1. Motivation

Recall that in Figure 1, after we collected SYN-sequence Q_0, we derived three race variants, namely V_1, V_2, and V_3. Variant V_1 was derived by changing the race outcome of r_3, V_2 was derived by changing the race outcome of r_1, and V_3 was derived by changing both of these race outcomes. In general, when we derive the race variants of a SYN-sequence, we need to derive one variant for each valid combination of race outcome changes. Intuitively, a valid combination of race outcome changes is one that can be used to derive a race variant. Assume that a SYN-sequence has m receive events r_i, each having d_i different race outcomes, where $1 \leq i \leq m$. Then, the number of valid combinations of race outcome changes is on the order of $d_1 \times \cdots \times d_m$. (As discussed later, some combinations are not valid and cannot be used to derive race variants.) Generating one race variant for each valid combination, and doing so for each SYN-sequence that is exercised, maximizes test coverage, but is impractical for many applications. A natural question to ask is: Can we derive a smaller number of race variants, and thus exercise a smaller number of SYN-sequences, but still effectively detect faults?

In this section, we describe a new testing strategy, called t-way reachability testing, that adopts the dynamic framework of reachability testing but uses t-way testing to selectively exercise a set of race variants for each SYN-sequence. Assume that a system has n parameters. Instead of covering all the n-way combinations, i.e. combinations involving all the n parameters, t-way testing only covers all the t-way combinations, where t is smaller than n. A key observation behind t-way testing is that not every parameter contributes to every fault in a system, and many faults can be exposed by considering interactions among a small number of parameters. To illustrate the concept of t-way testing, consider a system consisting of three Boolean parameters. Figure 2 shows a 2-way (or pairwise) test set for this system. Each row represents a test, and each column represents a parameter. It is easy to check that any two columns from the test set contain all four combinations, $\{00, 01, 10, 11\}$ of any two of the Boolean parameters. Note that if all the faults are caused by interactions between two parameters, this pairwise test set is able to detect all the faults in the system. Also note that the exhaustive test set for this system consists of eight tests, one for each combination of values of the three Boolean parameters.

Intuitively, t-way reachability testing considers each receive event as a parameter, and each possible race outcome change that can be made to a receive event as a value of the parameter representing the receive event. Then, when we derive race variants, instead of deriving a race variant for every valid combination of race outcome changes, we derive a set of race variants to only cover all the t-way valid combinations of race outcome changes. For example, in Figure 1, there is a total of two valid race outcome changes that can be made in SYN-sequence Q_0. One is to

$$\begin{pmatrix} 0\,0\,0 \\ 0\,1\,1 \\ 1\,0\,1 \\ 1\,1\,0 \end{pmatrix}$$

Figure 2. An example pairwise.

DOI: 10.1002/stvr

change the send partner of receive event r_1 from s_1 to s_2; the other is to change the send partner of receive event r_3 from s_3 to s_4. Both changes are covered in V_3, and thus deriving variant V_3 would be sufficient for 1-way testing. Note that the first change is covered in variant V_1, and the second change is covered in variant V_2. Thus, variants V_1 and V_2 together also form a 1-way test set. However, an optimizing algorithm would choose variant V_3. Note that all the three variants together form a 2-way test set, which is also the exhaustive test set, for sequence Q_0.

4.2. Computing race variants

A key component of the reachability testing framework is the computation of race variants. This component is also where t-way reachability testing differs from exhaustive reachability testing. In this section, we discuss how the race variants are derived in an exhaustive reachability testing algorithm. In the next section, we will describe the changes that are needed for t-way reachability testing.

One approach to deriving the race variants of a SYN-sequence Q is to build a 'race table' for Q. Each row of the race table represents, and is labelled by, a unique, partially ordered race variant of Q. In the race table, there is a column for each receive event in Q whose race set is non-empty. As an example, Table I shows the race table built for SYN-sequence Q_0 in Figure 1. In Q_0, $race\_set(r_1) = \{s_2\}$, $race\_set(r2) = \{\}$, $race\_set(r_3) = \{s_4\}$, and $race\_set(r4) = \{\}$. Thus, Table I has columns for receive events r_1 and r_3. Variants V_1, V_2, and V_3 in Figure 1 are derived from rows 1, 2, and 3, respectively, of Table I, as described below.

Let r be a receive event represented by one of the columns and V be a race variant represented by one of the rows. The value v in the row for V and the column for r indicates how receive event r in sequence Q is changed to create variant V:

- $v = -1$ indicates that r is removed from V.
- $v = 0$ indicates that no new send partner is specified for r in V.
- $v > 0$ indicates that in V, the send partner of r is changed to the vth event in $race\_set(r)$, where the send events in $race\_set(r)$ are arranged in an arbitrary order and the index of the first event in $race\_set(r)$ is 1.

Recall from Section 3 that whenever we change the send partner of a receive event r, we must remove any event whose existence possibly depends on r. The events that an event e depends on can be determined by computing the *control structure* of e. Let e be an event exercised by a thread T in a SYN-sequence Q. Then, the control structure of e in Q, denoted as *c-struct*(e, Q) *or c-struct*(e) if Q is implied, is empty if e is the first event exercised by T; otherwise, it is the prefix of Q that contains the event f that T exercised immediately before e, plus all the events that *happened before* f, including the synchronizations between these events. Note that the happened-before relation

Table I. Race table for sequence Q_0 in Figure 1.

Race variant	r_1	r_3
V_1	0	1
V_2	1	0
V_3	1	1

DOI: 10.1002/stvr

is the usual one defined in [32]. Intuitively, an event e_1 happened before another event e_2 in a SYN-sequence Q if e_1 could potentially affect e_2. In a space–time diagram, e_1 could potentially affect e_2 if information could flow from e_1 to e_2 following the vertical lines (from top to bottom) and the arrows. For example, in Figure 1, the control structure of r_2 in Q_0 contains s_1 and r_1. Note that s_2 happened before r_2, but s_2 is not in the control structure of r_2 since s_2 did not happen before r_1, where r_1 is the event T_2 exercised immediately before exercising r_2.

In [5], we reported an efficient algorithm for constructing the race table of a sequence Q. The algorithm generates all valid combinations of race outcome changes and adds each of them as a row to the resulting race table. Denote the individual values in a combination c as $c[1], c[2], \ldots, c[n]$. Then, c is valid if all of the following rules are satisfied:

Rule 1: There exists at least one value $c[i]$, $1 \leq i \leq n$, such that $c[i]>0$. This rule ensures that at least one race outcome change is made in SYN-sequence Q to derive a variant.

Rule 2: $c[i]=-1$, $1 \leq i \leq n$, if and only if there exists an index j, where $1 \leq j \leq n$ and $j \neq i$, such that $c[j]>0$ and $r_j \in c\text{-}struct(r_i)$. This rule ensures that a receive event r_i is removed from SYN-sequence Q if and only if the race outcome of another receive event r_j in the control structure of r_i is changed.

Rule 3: If $c[i]>0$, there does not exist an index j, $1 \leq j \leq n$, such that $c[j]>0$ and $r_j \in c\text{-}struct(s)$, where s is the $c[i]$th send event in $race\_set(r_i)$. This rule ensures that the send partners of the receive events exist in the race variant, i.e. that the send partners are not removed due to a change in the race outcome of some receive event that happened before them.

Note that the above rules ensure that each combination added into the race table can indeed be used to derive a race variant. In the next section, we present a race table construction algorithm that covers all valid t-way combinations, instead of all valid combinations, of race outcome changes. The main challenge of the new algorithm is to cover all the t-way combinations with a race table that satisfies rules 1, 2, and 3, and that is as small as possible.

4.3. Algorithm CombinatorialRT

Figure 3 shows an algorithm, named *CombinatorialRT*, which implements our combinatorial testing strategy. Algorithm *CombinatorialRT* starts with the SYN-sequence Q_0 collected during a non-deterministic test run (line 1). Recall that a non-deterministic test run is a test run in which race conditions are resolved arbitrarily, i.e. without controlling which SYN-sequence gets exercised. *CombinatorialRT* then calls function *ConstructRaceTable* in Figure 4 to construct a race table of Q_0. This race table is used to derive a set $variants(Q_0)$ of Q_0 (line 2) and set $variants(Q_0)$ is used to initialize set *variants* (line 3). The while-loop (lines 4–8) first checks whether or not set *variants* is empty (line 4). If not, each variant in set *variants* is used to collect a new SYN-sequence by performing prefix-based testing (lines 5 and 6). Prefix-based testing with a race variant deterministically forces the race variant to be exercised, and then allows the execution to proceed non-deterministically. Each newly collected SYN-sequence Q is then used to derive new race variants (line 7), which are added to set *variants* (line 8). The while-loop iterates until set *variants* becomes empty.

Note that if we apply algorithm *CombinatorialRT* multiple times to the same program with the same input, different sets of SYN-sequences could be exercised. The reason is that a portion of a prefix-based test run is non-deterministic. Also note that algorithm *CombinatorialRT* shares the exact

DOI: 10.1002/stvr

```
ALGORITHM CombinatorialRT(CP: a concurrent program; X: an input of CP; t: degree
of interaction){
1. collect a SYN-sequence Q0 by executing CP with input X non-deterministically;
2. construct a race table of Q0 by calling function ConstructRaceTable with Q0 and t
     and derive a set of variants(Q0) of race variants, one from each row of the race
     table
3. variants = variants (Q0);
4. while (variants is not empty) {
5.   withdraw a variant V from variants;
6.   collect a SYN-sequence Q by conducting a prefix-based test run with V;
7.   construct a race table of Q by calling function ConstructRaceTable with Q and t
       and derive a set of variants(Q) of race variants, one from each row of the race
       table
8. variants = variants ∪ variants(Q);
   }
}
```

Figure 3. Algorithm *CombinatorialRT*.

```
FUNCTION ConstructRaceTable (Q: a SYN-sequence, t: degree of interaction) {
1. initialize table = (heading, matrix) to be an empty race table
2. let R be the set of receive events in Q whose race sets are non-empty
3. let heading = (r1, r2, ..., r|R|) be a topologic order of R w.r.t. the happened-before
   relation
4. let D = {d1, d2, ..., d|R|}, where di = |race_set(r)|
5. add each t-way combination τ = (r1.v1, r2.v2, ..., rt.vt), where -1≤ vi≤ di, into matrix as
   a row if τ satisfies rules 1, 2 and 3
6.  for (int i = t + 1;i <= |R|; i ++) {
7.    let π be the set of t-way combinations (rk1.vk1, rk2.vk2, ...,rkt-1. vkt-1,ri.vi),
        where 1≤ k1 < k2 < ... < kt-1≤ i -1 and -1≤ vki≤ di, that satisfy rules 1, 2 and 3
      // horizontal growth for receive event ri
8.    for (each row τ = (r1.v1, r2.v2, ..., ri-1.vi-1) in matrix)  {
9.      replace τ with (r1.v1, r2.v2, ..., ri-1.vi-1, ri.vi) such that it satisfies rules 2 and 3
          and covers the most number of combinations in π
10.     remove the t-way combinations covered by τ from π
      }
      // vertical growth for receive event ri
11.   for (each combination σ in π){
12.     if (there exists a row τ in matrix such that it can be changed to cover σ
            without violating rules 2 and 3)
13.       change τ so that it covers combination σ
        }
14.     else {
15.       add a new row to cover combination σ
        }
16.     remove σ from π
      }
    }
17. return table;
}
```

Figure 4. Function *ConstructRaceTable*.

DOI: 10.1002/stvr

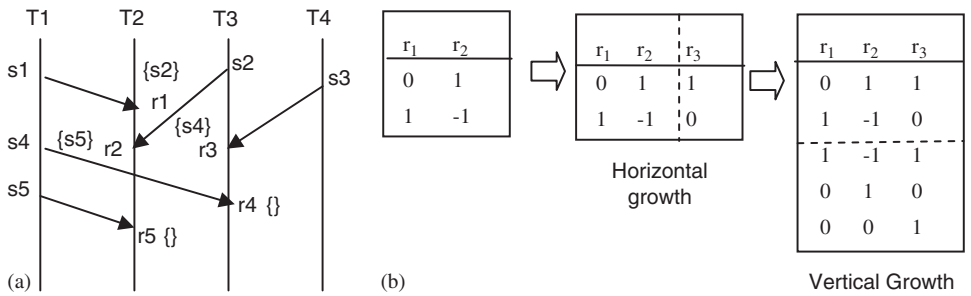

Figure 5. Illustration of function *ConstructRaceTable*: (a) an example SYN-sequence Q and (b) a race table for sequence Q.

same framework as the exhaustive reachability testing algorithm reported in [5]. To simplify the presentation, we have omitted some details about how this framework ensures that the reachability testing process terminates and that no SYN-sequence is exercised more than once. These details are conceptually independent from, and are not affected by, the introduction of t-way testing into the reachability testing process.

Given an integer t and a SYN-sequence Q, function *ConstructRaceTable* in Figure 4 adapts the IPO strategy to build a t-way race table for Q. To understand this function, it helps to consider each receive event as a parameter, and the set of values a receive event r (or more precisely, the parameter representing event r) can take is $\{-1, 0, \ldots, |race\_set(r)|\}$. The framework of function *ConstructRaceTable* is as follows: it builds a t-way race table for the first t receive events, extends the race table to build a t-way race table for the first $t+1$ receive events, and continues to do so until it builds a t-way race table for all the receive events in Q. Figure 5 shows an example SYN-sequence and a pairwise race table constructed by function *ConstructRaceTable*. We will use this as a running example to illustrate how function *ConstructRaceTable* works. Note that in the example SYN-sequence, the race set of each receive event is identified inside a pair of curly braces next to the receive event.

Function *ConstructRaceTable* first places the receive events with non-empty race sets into a topological order with respect to the happened-before relation (line 3). (If receive event a happened before receive event b, then a appears to the left of b.) Note that receive events with empty race sets are excluded because the race outcomes of those receive events cannot be changed. In Figure 5, three receive events, namely r_1, r_2, and r_3, have non-empty race sets. The happened-before relation between the three events is: r_1 happened before r_2, r_1 is concurrent with r_3, and r_2 is concurrent with r_3. Thus, one topologic ordering of these events has r_1 appearing first, then r_2, and then r_3, as shown in the heading of the race table in Figure 5(b).

Next, a t-way race table for the first t receive events is built (line 5). This race table simply contains all the valid combinations of race outcome changes that can be made to the first t receive events. A combination of race outcome changes is considered to be valid if it satisfies rules 1, 2, and 3. In Figure 5, the pairwise race table for the first two receive events r_1 and r_2 consists of two valid combinations, namely $\{r_1.0, r_2.1\}$, and $\{r_1.1, r_2.-1\}$. These two combinations are the only combinations of race outcome changes that can be made to r_1 and r_2 that satisfy rules 1, 2, and 3. Here we have used the notation $r.v$ to represent the race-set value v of receive event r. Note that

DOI: 10.1002/stvr

in the combination $\{r_1.1, r_2.-1\}$, the race-set value of r_2 is -1 since the send partner of r_1 has been changed and r_1 happens before r_2. More details about the enforcement of rules 1, 2, and 3 are provided later.

The remaining receive events are covered, one at a time, by the outermost for-loop (line 6) as described below. Let r_i be the receive event being covered by the current iteration. The extension of the race table for receive event r_i first computes the set π of combinations that need to be covered. Note that all the receive events $r_1, \ldots, r_{i-1}$ have already been covered. Thus, in order to cover receive event r_i, we only need to cover all valid t-way combinations involving r_i and each of the possible groups of $t-1$ receive events among those that are already covered (line 7). In Figure 5, in order to cover r_3, we only need to cover all the valid 2-way combinations involving r_3 and r_1 and all the valid 2-way combinations involving r_3 and r_2. There are six such combinations, i.e. $\pi = \{(r_1.0, r_3.1), (r_1.1, r_3.0), (r_1.1, r_3.1), (r_2.0, r_3.1), (r_2.1, r_3.0), (r_2.1, r_3.1)\}$.

The combinations in set π are covered in the following two steps:

- *Horizontal growth*: This step extends each of the existing rows by adding a value for the new receive event (lines 8–10). These values are chosen in a greedy manner, i.e. at each step, the value chosen is the one that covers the largest number of combinations in π (line 9). In Figure 5, the first row is extended by adding the value 1 for r_3, which covers two combinations $\{(r_1.0, r_3.1), (r_2.1, r_3.1)\}$. Note that if the first row was extended by adding the value 0, it would only cover one valid combination $(r_2.1, r_3.0)$. The second row is extended by adding the value 0 for r3, which covers one valid combination $(r_1.1, r_3.0)$. Note that if the second row was extended by adding the value 1, it would also cover only one valid combination $(r_1.1, r_3.1)$. In this case, the tie is arbitrarily broken in favour of 0.
- *Vertical growth*: This step covers the remaining uncovered combinations, one at a time, either by changing an existing row (line 13) or by adding a new row (line 15). Note that when we change an existing row to cover a combination, only *don't care* values can be changed and these changes must satisfy rules 2 and 3. (The reason why rule 1 is not enforced here will be explained later.) Also note that when we add a new row to cover a combination σ, the receive events involved in σ are assigned the same values as in σ, and the other receive events are assigned *don't care* values or appropriate values so that the new row satisfies rules 2 and 3. In Figure 5, after horizontal growth, three combinations have yet to be covered: $\{(r_1.1, r_3.1), (r_2.1, r_3.0), (r_2.0, r_3.1)\}$. To cover $(r_1.1, r_3.1)$, we add row $(r_1.1, r_2.-1, r_3.1)$. Note that in this row, the value of r_2 is -1, since r_1 happens before r_2, and the race outcome of $r_1$is changed. This means that r_2 will be removed when we derive a race variant from this row. To cover $(r_2.1, r_3.0)$ we add row $(r_1.0, r_2.1, r_3.0)$, and to cover $(r_2.0, r_3.1)$ we add row $(r_1.0, r_2.0, r_3.1)$. Note that in both cases, the value of r_1 has to be 0 because otherwise the value of r_2 would have to be -1 (since r_1 happens before r_2).

Now we discuss how rules 1, 2, and 3 are enforced in function *ConstructRaceTable*. Rules 1, 2, and 3 are enforced when we compute the set of missing combinations (lines 5 and 7) and rules 2 and 3 are enforced when we add a new value or change an existing value in the race table (lines 9 and 12). The former ensures that only valid t-way combinations are covered. The latter ensures that each row in the race table can be used to derive a race variant. Note that rule 1 is automatically enforced in the latter case. The reason is that rows are added in lines 5 and 15 and in both lines

DOI: 10.1002/stvr

rows are added to cover an uncovered combination that satisfy rule 1. Thus, all the rows in the race table must satisfy rule 1 as well. Also note that these rules need to be enforced each time a new value is added, which can be accomplished by comparing the newly added value to each of the existing values in the same row. Thus, the complexity of enforcing these rules is linear with respect to the number of values in each row, which is less than or equal to the number of receive events in a SYN-sequence.

We point out that algorithm *CombinatorialRT* is largely a combination of the exhaustive reachability testing algorithm in [5] and the IPO-based t-way test generation algorithm in [23]. Note, however, that the original IPO-based algorithm in [23] assumes that every combination is valid and thus does not provide any support for enforcing the rules that are required for constructing t-way race tables. Also note that other t-way testing strategies, e.g. AETG [19], could be adapted similarly to be used in algorithm *CombinatorialRT*.

5. EXPERIMENTAL RESULTS

The goal of our experiments was to assess the effectiveness of the t-way reachability testing strategy, both in terms of its effectiveness for detecting errors and the reduction in the number of sequences that are exercised during t-way reachability testing, as compared to exhaustive reachability testing.

In [5], we reported an exhaustive reachability testing tool called RichTest. In our experiments, RichTest was extended to implement our t-way reachability testing algorithm, namely algorithm *CombinatorialRT*. RichTest is developed in Java, and consists of three main components: a synchronization library, a race variant generator class, and a test driver class. The synchronization library provides classes for semaphores, monitors, and message passing with selective waits. These synchronization classes contain the necessary control for tracing SYN-sequences and replaying variants. The race variant generator is responsible for constructing the race table and for deriving the race variants. The test driver is responsible for coordinating the exchange of variants and SYN-sequences between the synchronization classes and the variant generator. These three components and the application form a single Java program that performs the reachability testing process.

As a proof-of-concept, we conducted an empirical study in which RichTest was used to perform t-way reachability testing on several programs. The programs used as objects of the study include:[§]

- DDP: a solution to the distributed dining philosophers problem [33] with three processes.
- DME: a solution to the distributed mutual exclusion problem [34] with three processes.
- TDME: a token-based solution to the distributed mutual exclusion problem [35] with three processes.
- DST-C: a solution to the distributed spanning tree problem [33], which computes a spanning tree for a *complete* graph of four nodes. Program DST-C had four processes, one per node.
- DST-I: a solution to the distributed spanning tree problem [33], which computes a spanning tree for an *incomplete* graph of four nodes. Program DST-I had four processes, one per node.

[§] These programs can be obtained by contacting the authors.

DOI: 10.1002/stvr

Each process in programs DDP, DME, and TDME has two threads. Message passing is used for communication between the processes in these programs, and semaphores are used for synchronization between the threads within a single process. Each process in programs DST-C and DST-I consists of a single thread.

Note that these solutions represent synchronization/communication patterns that are commonly found in many practical applications. Also note that the synchronization behaviours of all the above programs are independent from their inputs. We stress that the complexity of the synchronization behaviour of a program mainly depends on the way in which the threads communicate and synchronize, not the size of the program.

We measured the adequacy of the test sequences generated during t-way reachability testing by using mutation testing. For each of the five programs described above, we generated a batch of mutants using the Java-based mutation tool μJava [36,37]. A mutant introduces a single change to the original program and is intended to simulate a programming error. We used all of the method-level mutants generated by the μJava tool. The method-level (traditional) mutants are based on the selective operator set defined in [38]. Some of the mutants created were functionally equivalent to the original program. These mutants were identified and deleted. We then applied t-way reachability testing to each of the non-equivalent mutants. A mutant was considered to be *killed* if an execution of the mutant caused a runtime error or failed a correctness check that was performed during each execution of t-way reachability testing. As an example of a correctness requirement, we checked that every process in the DME program entered its critical section and that no two processes were in their critical sections at the same time. A mutant that was not killed by any test run during t-way reachability testing was said to be *alive* after testing. The number of mutants that are alive after testing is a measure of the adequacy of the test sequences. Note that the less the mutants are alive after testing, the more effective t-way reachability testing is.

Table II summarizes the results of t-way reachability testing for all the programs. For each program, we show the number of non-equivalent mutants, the number of mutants that were alive after 1-way reachability testing, and the number of mutants that were alive after 2-way reachability testing. As Table II shows, 1-way reachability testing killed all the mutants for all the programs except DME. For program DME, 1 mutant was alive after 1-way reachability testing. Note that 2-way reachability testing killed all the mutants for all the programs.

Table III shows the difference between the total number of sequences exercised during exhaustive reachability testing and the number of sequences exercised during 1-way and 2-way reachability testing. Since the number of sequences exercised during an execution of 1-way or 2-way reachability testing is non-deterministic, we performed reachability testing 5 times for each program. Table III reports the average number of sequences exercised for each program. For all programs except TDME, 1-way testing substantially reduced the number of sequences exercised as compared to exhaustive reachability testing. Yet, as Table II showed, 1-way testing was still effective at detecting faults. The reduction for 2-way testing was substantial for programs DDP, DME, and DST-C. Note that the number of sequences that can possibly be exercised by a concurrent program can be used to measure the complexity of the program. The reduction results suggest that 1- and 2-way testing can achieve a greater reduction for more complex programs. We point out that the amount of reduction will grow as the number of processes increases. For example, DME with four processes can exercise over 70 million possible sequences, but 1-way testing exercises only 12 607 sequences on average.

DOI: 10.1002/stvr

Table II. Results of t-way reachability testing.

Program	Mutants	Mutants alive after 1-way testing	Mutants alive after 2-way testing
DDP	303	0	0
DME	172	1	0
TDME	123	0	0
DST-C	199	0	0
DST-I	199	0	0

Table III. Number of sequences exercised during t-way reachability testing.

Program	Number of sequences exercised during exhaustive testing	Number of sequences exercised during 1-way testing	Number of sequences exercised exercised during 2-way testing
DDP	5 380 996	5199	1 172 057
DME	4032	183	1169
TDME	24	23	23
DST-C	25 200	79	2142
DST-I	30	8	29

Table IV. Two parameters affecting race table sizes.

	Number of recvs with non-empty race sets (per sequence)			Average race set size (per sequence)	
Program	Max	Min	Average	Max	Min
DDP	19	1	1.74	1.71	1
DME	12	1	1.92	1	1
TDME	4	1	1.92	1	1
DST-C	13	1	1.75	1.67	1
DST-I	4	1	1.57	1.5	1

Table IV shows some statistics on two parameters that directly affect the race table sizes. The first parameter, i.e. the number of receive events with non-empty race sets, is essentially the number of columns in a race table. The second parameter, i.e. the average race set size, is essentially the average number of different values that could appear in a column. These statistics were collected during an application of exhaustive reachability testing to each subject program. We expect a greater reduction in the number of sequences exercised to be achieved for programs where the two parameters have bigger values.

Even though Tables II and III show that 1-way and 2-way testing were very effective at killing mutants, we expect that higher values of t will be required for some programs, especially as the number of processes increases. Note that the minimum number of processes needed for testing is usually not known and is difficult to determine. The reason is two-fold. First, we typically do not know what faults actually exist in an arbitrary program. Second, even if we target a particular type of faults, the minimum number of processes needed for detecting a fault usually depends on the

DOI: 10.1002/stvr

program logic, and is not known a *priori*. Some recent work on computing this 'cutoff number' has been reported. Intuitively, the cutoff number C of processes for testing a concurrent program CP is defined with respect to a certain type T of faults, e.g. deadlock, such that if CP has no fault of type T with C processes, then CP will have no fault of type T with any number of processes. More information about the 'cutoff number' can be found in [39–41].

Finally, we discuss the threats to the validity of our experiments. There are two main threats to external validity. First, the concurrent programs studied in our experiments implement solutions to several classical concurrency problems. These solutions represent synchronization patterns that are found in many programs. However, the programs in our experiments may not be representative, in terms of scale and complexity, of industrial applications. This threat can be reduced by conducting experiments on additional programs that represent other synchronization patterns and on industrial applications.

The second threat to external validity is that the seeded faults may not be representative of real faults. To reduce this threat, the faults in our experiments were seeded using a third-party mutation tool, namely μJava, which ensured that a wide variety of faults were systematically inserted in an impartial way. Mutation testing has been shown to be an effective tool for the empirical assessment of testing techniques [42]. Note that even though μJava was designed for mutation testing of sequential programs, μJava can be effectively applied to concurrent programs. This is because many programming errors in sequential programs also exist in concurrent programs and can cause synchronization faults. For example, one common programming error in a sequential program is writing an incorrect Boolean operator in a Boolean expression. This error would cause a synchronization fault in a concurrent program if such a Boolean expression was used to decide whether or not a synchronization operation should be performed. The threat associated with fault seeding can be further reduced by using a mutation testing tool that is specially designed for concurrent programs, which could cover synchronization faults that are currently missing in μJava. To the best of our knowledge, such a tool is yet to be developed.

The main threat to internal validity is that our prototype tool may not correctly implement the t-way reachability testing algorithm. To reduce this threat, a coverage checker was implemented and used to ensure that each race table built by our tool satisfied t-way coverage.

6. CONCLUSION AND FUTURE WORK

Combinatorial testing has been shown to be effective for general software applications. However, it has not been used for testing concurrent programs. In this paper we have presented a combinatorial testing strategy for concurrent programs. This strategy is based on an existing technique called reachability testing. Reachability testing has the advantage of being able to derive test sequences automatically and on-the-fly, without constructing a static model. However, existing reachability testing techniques are exhaustive, which is impractical for many applications. To enable a balance between test coverage and test effort, our new strategy uses a combinatorial strategy called t-way testing to selectively exercise the test sequences that are derived during reachability testing. The main idea is to cover all the t-way combinations, instead of *all* the combinations, of the changes that can be made to the race outcomes in a test sequence. We have presented an algorithm that implements the new strategy, and have reported an empirical study to assess the effectiveness of

DOI: 10.1002/stvr

this new strategy. The results of the study indicate that the new strategy can substantially reduce the number of test sequences that need to be exercised while still effectively detecting faults.

We will continue our work in the following directions. First, we plan to conduct more case studies to assess the effectiveness of t-way reachability testing. In particular, we plan to identify some industrial and/or open-source applications and apply t-way reachability testing to these applications. Second, we plan to compare t-way reachability testing to reachability testing techniques that are based on global coverage criteria, both in terms of their ability to detect faults and the number of SYN-sequences that need to be exercised. Examples of global coverage criteria include covering all branches, all synchronizable pairs, and all definition-use pairs. This is different from t-way reachability testing, which selects test sequences *locally* in the sense that the t-way testing strategy is applied to individual test sequences. Finally, we plan to develop a graphical user interface (or GUI) for our reachability testing tool. The GUI will make it easier for the user to specify correctness properties and to inspect the test runs.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers, whose comments have greatly improved this paper. This work is partly supported by a grant (Award No. 60NANB6D6192) from the Information Technology Lab (ITL) of National Institute of Standards and Technology (NIST).

Disclaimer

Certain software products are identified in this document. Such identification does not imply recommendation by NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

REFERENCES

1. Tai KC. Testing of concurrent software. *Proceedings of the 13th Annual International Computer Software and Applications Conference* (*COMPSAC*), 1989; 62–64.
2. Ramalingam G. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems* 2000; **22**(2):416–430.
3. Carver R, Lei Y. A general model for reachability testing of concurrent programs. *Proceedings of International Conference on Formal Engineering Methods*, 2004; 76–98.
4. Hwang GH, Tai KC, Huang TL. Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering* 1995; **5**(4):493–510.
5. Lei Y, Carver RH. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering* 2006; **32**(6):382–403.
6. Lei Y, Wong E. A novel framework for non-deterministic testing of message-passing programs. *Proceedings of IEEE International Conference on High Assurance System Engineering* (*HASE*), 2005; 66–75.
7. Edelstein O, Farchi E, Nir Y, Ratsaby G, Ur S. Multithread Java program test generation. *IBM Systems Journal* 2002; **41**(1):111–125.
8. Stoller SD. Testing concurrent Java programs using randomized scheduling. *Proceedings of the Second Workshop on Runtime Verification* (*RV*), *also appears in Electronic Notes in Theoretical Computer Science*, vol. 70, no. 4. Elsevier: Amsterdam, 2002; 1–16.
9. Yang RD, Chung CG. A path analysis approach to concurrent program testing. *Information and Software Technology* 1992; **34**(1):43–56.
10. Yang C, Souter AL, Pollock LL. All-du-path coverage for parallel programs. *International Symposium on Software Testing and Analysis* (*ISSTA*), 1998; 153–162.

DOI: 10.1002/stvr

11. Taylor RN, Levine DL, Kelly CD. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering* 1992; **18**(3):206–214.
12. Corbett J, Dwyer M, Hatcliff J, Laubach S, Pasareanu C, Robby, Zheng H. Bandera: Extracting finite-state models from Java source code. *Proceedings of International Conference on Software Engineering*, June 2000; 439–448.
13. Havelund K, Pressburger T. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer* (*STTT*) 2000; **2**(4):366–381.
14. Godefroid P. Software model checking: The VeriSoft approach. *Formal Methods in System Design* 2005; **26**(2):77–101.
15. Nilsson R, Offutt J, Andler SF. Mutation-based testing criteria for timeliness. *Proceedings of the 28th IEEE Annual Computer Software and Applications Conference* (*COMPSAC*), 2004; 306–312.
16. Nilsson R, Offutt J, Mellin J. Test case generation for mutation-based testing of timeliness. *Electronic Notes in Theoretical Computer Science*. Springer: Berlin, 2006; 102–121.
17. Pettersson A, Thane H. Testing of multi-tasking real-time systems with critical sections. *Proceedings of Ninth International Conference on Real-Time Computing Systems and Applications* (*RTCSA*), 2003; 578–594.
18. Thane H. Monitoring, testing and debugging of distributed real-time systems. *Ph.D. Thesis*, Royal Institute of Technology, KTH, Stockholm, Sweden, 2000.
19. Cohen DM, Dalal SR, Fredman ML, Patton GC. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 1997; **23**(7):437–444.
20. Colbourn CJ, Cohen MB, Turban RC. A deterministic density algorithm for pairwise interaction coverage. *Proceedings of International Conference on Software Engineering* (*SE*), 2004; 245–252.
21. Tung YW, Aldiwan WS. Automating test case generation for the new generation mission software system. *Proceedings of IEEE Aerospace Conference*, 2000; 431–437.
22. Bryce R, Colbourn CJ, Cohen MB. A framework of greedy methods for constructing interaction tests. *Proceedings of IEEE International Conference on Software Engineering* (*ICSE*), May 2005; 146–155.
23. Tai KC, Lei Y. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering* 2002; **28**(1):109–111.
24. Lei Y, Tai KC. In-parameter-order: A test generation strategy for pairwise testing. *Proceedings of IEEE International High-Assurance Systems Engineering Symposium*, 1998; 254–261.
25. Cohen MB, Colbourn CJ, Gibbons PB, Mugridge WB. Constructing test suites for interaction testing. *Proceedings of IEEE International Conference on Software Engineering* (*ICSE*), 2003; 38–48.
26. Burr K, Young W. Combinatorial test techniques: Table-based automation, test generation and code coverage. *Proceedings of International Conference on Software Testing Analysis & Review*, 1998; 503–513.
27. Dalal SR, Jain A, Karunanithi N, Leaton JM, Lott CM, Patton GC, Horowitz BM. Model-based testing in practice. *Proceedings of International Conference on Software Engineering* (*ICSE*), 1999; 285–294.
28. Kuhn R, Wallace D, Gallo A. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering* 2004; **30**(6):418–421.
29. Grindal M, Lindström B, Offutt J, Andler SF. An evaluation of combination testing strategies. *Empirical Software Engineering* 2006; **11**(4):583–611.
30. Ammann PE, Offutt AJ. Using formal methods to derive test frames in category-partition testing. *Proceedings of the Ninth Annual Conference on Computer Assurance* (*COMPASS*), June 1994; 69–80.
31. Andrews GR. *Foundations of Multithreaded*, *Parallel*, *and Distributed Programming*. Addison-Wesley: Reading, MA, 2000.
32. Lamport L. Time, clocks, and the ordering of events in a distributed system. *Communication of ACM* 1978; **21**(7):558–565.
33. Garg V. *Concurrent and Distributed Computing in Java*. Wiley: New York, 2004.
34. Ricart G, Agrawala AK. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM* 1981; **24**(1):9–17.
35. Suzuki I, Kasami T. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems* 1985; **3**(4): 344–349.
36. Ma Y, Offutt J, Kwon Y. MuJava: An automated class mutation system. *Journal of Software Testing*, *Verification and Reliability* 2005; **15**(2):97–133.
37. μJava Home Page, http://ise.gmu.edu/∼ofut/mujava/
38. Offutt J, Lee A, Rothermel G, Untch R, Zapf C. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology* 1996; **5**(2):99–118.
39. Zhou J, Tai KC. Deadlock analysis of synchronous message-passing programs. *Proceedings of International Symposium on Software Engineering for Parallel and Distributed Systems*, 1999; 62–71.
40. Zhou J, Tai KC. Efficient deadlock analysis of clients/server systems with two-way communication. *Proceedings of International Symposium on Software Reliability Engineering*, 2001; 222–231.
41. Zhou J, Tai KC. Deadlock analysis of client/server programs. *Proceedings of International Conference on Distributed Computing Systems*, 2000; 484–491.
42. Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? *Proceedings of IEEE International Conference on Software Engineering* (*ICSE*), May 2005; 402–411.

DOI: 10.1002/stvr