SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2015; **25**:310–332
Published online 9 February 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1568

Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs

S. R. S. Souza[1,*,†], P. S. L. Souza[1], M. A. S. Brito[1], A. S. Simao[1] and E. J. Zaluska[2]

[1]*Computer Systems Department, University of Sao Paulo, Sao Carlos, 13566-590, Brazil*
[2]*Electronic and Computer Science, University of Southampton, Southampton, SO17 1BJ, UK*

SUMMARY

Testing is a key activity to assure the quality of concurrent applications. In recent years, a variety of different mechanisms have been proposed to test concurrent software. However, a persistent problem is the high testing cost because of the large number of different synchronization sequences that must be tested. When structural testing criteria are adopted, a large number of infeasible synchronization sequences is generated, increasing the testing cost. Although the use of reachability testing reduces the number of infeasible combination (because only feasible synchronization sequences are generated), many synchronization combinations are also generated, and this again results in a testing cost with exponential behavior. This paper presents a new composite approach that uses reachability testing to guide the selection of the synchronization sequences tests according to a specific structural testing criterion. This new composite approach is empirically evaluated in the context of message-passing concurrent programs developed with MPI. The experimental study evaluates both the cost and effectiveness of proposed composite approach in comparison with traditional reachability testing and structural testing. The results confirm that the use of the new composite approach has advantages for testing of concurrent applications. Copyright © 2015 John Wiley & Sons, Ltd.

Received 15 April 2013; Revised 17 November 2014; Accepted 11 January 2015

KEY WORDS: structural testing; reachability testing; concurrent programs; experimental study

1. INTRODUCTION

Concurrent software is being increasingly used with the availability of multicore processors and computer clusters. Many modern business applications now use concurrency to improve overall system performance. However, all concurrent software contains features, such as nondeterminism, synchronization, and interprocess communication, which make concurrent software significantly more complex than sequential software. These features impose new challenges for testing and increase the difficulty of demonstrating the correct execution of the application in all possible conditions.

Traditional testing techniques are often not well-suited to the testing of concurrent (or parallel) software. Many researchers have developed specific testing techniques to address specific issues such as nondeterminism, concurrency, communication, synchronization, race conditions, and replay testing [1–11].

In previous work, structural testing criteria for concurrent programs were proposed by Souza *et al.* [12], defining test models and tools to test message-passing software. This work was extended by Sarmanho *et al.* [13] to test shared-memory software. These structural testing criteria are designed to explore information about the control, data, and communication flows of concurrent programs, considering both their sequential and parallel aspects. These contributions are related to

*Correspondence to: Souza, Simone Senger, USP/ICMC/SSC Avenida Trabalhador Sao-Carlense, 400, Centro, P.O.Box: 668, Sao Carlos, SP, 13566-590, Brazil.
†E-mail: srocio@icmc.usp.br

the coverage analysis concept, in which a coverage model is first defined, and then the model is evaluated during the execution of the program to determine whether the coverage model is satisfied or not. Information about uncovered elements (such as commands, synchronizations, and paths) is generated and the tester determines if these elements should be executed using new test cases or whether they can be safely ignored. This happens if there is no possible set of values for all the parameters (i.e. input and global variables) that cover that element. Therefore, this element is considered an *infeasible element*. A key objective of the proposed approach is to reduce the computation cost of testing by eliminating tests on infeasible elements whenever possible. However, the complete determination of all infeasible elements is not just extremely difficult in practice; moreover, this problem is actually theoretically impossible to solve (it is equivalent to the halting problem).

Coverage analysis improves the testing process by providing an efficient metric to evaluate testing activity progress and the quality of test cases. The approach uses a static analysis of the program under test to extract relevant *testing metadata*, which can then be used to generate the information required for fault coverage testing. Because of the static analysis, this testing assumes a conservative approach in which all receive events can be theoretically matched with all send events belonging to the other processes of the concurrent application (excluding receive and send events belonging to the same process). This approach has the advantage that it will identify any missing communication and thus improve the quality of the testing. However, a significant disadvantage is the resulting high testing cost because although some send–receive pairs can never occur in practice, they will still generate elements that must nevertheless be analysed by the tester, even though they are unnecessary (because they are infeasible). This disadvantage has been partially addressed by dynamic analysis approaches, in particular, reachability testing as explained in the succeeding text.

Reachability testing (in particular, the work of Lei and Carver [9]) is an approach that combines both nondeterministic and deterministic testing to generate automatically all sequences of synchronization events without constructing a static model. For the purposes of this paper, a synchronization event occurs when a match is established between a sender node (with a send primitive) and a receiver node (with a receive primitive) during the concurrent execution. Because of the nondeterminism inherent with receiver nodes, other (alternative) synchronization events could have been established with different sender nodes. In a general sense, reachability testing provides mechanisms for recovering all possible matches for a synchronization event and also forcing the testing of these matches. The approach is dynamic and only generates feasible send–receive pairs. Nevertheless, reachability testing generates a large number of possible combinations among synchronization events and, for complex programs, the analysis of this large number of potential synchronization events is computationally expensive.

It transpires that Lei and Carver's approach [9] is essentially complementary to the coverage testing criteria. Lei and Carver do not address the selection of the test case, which will be used for the initial run, whereas, for the coverage testing, static analysis of the program is employed to select the test cases in advance. Therefore, in Souza *et al.* [14] is proposed a new composite approach, using reachability testing to target coverage testing for synchronization events. The intention is to take appropriate advantage of both approaches: information about synchronization edges provided by the coverage criteria is then used to decide which synchronization events will be executed, selecting only ones not yet covered by existing test cases. It is possible to execute each synchronization at least once and use reachability testing to select only those feasible synchronizations to improve the testing coverage. Alternatively, the composite approach can be used to guide the selection of test cases to drive the reachability testing, i.e., information about synchronization coverage is used to select a particular test case for starting the reachability testing, potentially improving the overall results obtained. The approach of combining these two testing techniques has the potential to improve both the efficiency (i.e. there are a smaller number of tests) and the effectiveness (the ability to reveal faults) of concurrent programs testing.

This paper reports a controlled experiment, planned and conducted using the Experimental Software Engineering process, described by Wohlin *et al.* [15]. The experiment was conducted to evaluate the cost and the effectiveness of the composite approach, compared to both structural testing and reachability testing, in the context of message-passing concurrent programs developed with MPI (Message Passing Interface). To evaluate the effectiveness of the composite approach, defects were systematically inserted into the programs and the ability of the test set generated by this approach to detect these defects was evaluated. The results demonstrate that the composite approach is able to reduce the computational cost of the testing activity without reducing the ability to detect defects.

The paper is organized as follows: Section 2 describes reachability testing for concurrent programs; Section 3 addresses the previous work on coverage testing criteria for concurrent programs; Section 4 defines the composite testing approach; Section 5 describes the experimental study; Section 6 discusses related work and Section 7 reports the conclusions and future work.

2. REACHABILITY TESTING

The nondeterministic behaviour of concurrent programs renders them more difficult to test than sequential programs. An execution of a concurrent program nondeterministically exercises one out of many possible sequences of synchronization of send/receive events. Such a sequence is called a synchronization sequence (or *SYN-sequence*). One approach to test the concurrent program is to execute it many times, trying to exercise as many distinct SYN-sequences as possible, potentially producing different results, some of which can reveal faults. However, it is usually not possible to execute all possible SYN-sequences, due to the explosion of the number of synchronization events.

An alternative approach to deal with the nondeterministic behaviour is to compute all possible SYN-sequences and ensure that each one is executed, using a controlled execution. This approach removes the nondeterminism and the execution is now effectively deterministic. Note that most of the synchronization pairs are infeasible elements (as defined in Section 3) because of inevitable constraints of control and data flow in the program. Even though static analysis of the program can identify some of the infeasible synchronization pairs, in the general case it is not possible to even determine whether a given event (e.g., a receive primitive) will be executed, let alone determine with which particular send event it might synchronize.

Reachability testing was proposed as a dynamic approach for concurrent software testing capable of executing all feasible distinct SYN-sequences for a given input without using static analysis [16]. This approach requires the execution of the program in a semi-deterministic way; the execution deterministically follows the SYN-sequence up to a given point, after which it runs nondeterministically. Thus, given a (partial) SYN-sequence S_1, the program will eventually execute a SYN-sequence S_2 where S_1 is a prefix of S_2. The resulting SYN-sequence S_2 is analysed and any race conditions between the events are identified. Consider, for example, an SYN-sequence with a receive event r_i that synchronizes with the send event s_i, and a receive event r_j that synchronizes with another event s_j; there is a race condition between the synchronization pair. Thus, a new SYN-sequence can be obtained by matching r_i and s_j, and r_j and s_i. Such an SYN-sequence is called a *race variant* of the original SYN-sequence. In other words, a race variant of a SYN-sequence is a new SYN-sequence where a race condition between synchronization pairs is identified and altered. Each race variant contains one or more different synchronization from the original SYN-sequence and produces a new SYN-sequence that is executed following the reachability testing algorithm.

Reachability testing guarantees that all feasible synchronization will be exercised exactly once. The method provides a mechanism to deal with non-determinism, because all possible synchronizations for every test input are derived. The problem is the potentially high number of possible synchronization combinations generated and (for complex software) this number can be sufficiently

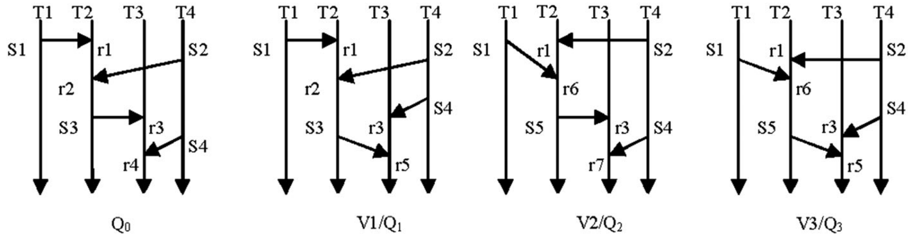

Figure 1. An example program and the possible combination of synchronization between threads.

high to restrict practical application. To illustrate this problem, consider the example in Figure 1 extracted from Lei and Carver [9]. The figure shows a space-time diagram in which vertical lines represent the execution of four threads $T1, T2, T3$ and $T4$ of a concurrent program. The interaction between processes is represented by arrows from a send event s to a receive event r. Diagram Q_0 shows the first execution of the program, generating synchronizations (s_1^{T1}, r_1^{T2}), (s_2^{T4}, r_2^{T2}), (s_3^{T2}, r_3^{T3}), and (s_4^{T4}, r_4^{T3}). V_1, V_2 and V_3 are *race variants* of Q_0 generated in the reachability testing. For instance, in V_1, the executed race variant is related to receiver r_3, resulting in the synchronizations (s_1^{T1}, r_1^{T2}), (s_2^{T4}, r_2^{T2}), (s_4^{T4}, r_3^{T3}) and (s_3^{T2}, r_5^{T3}). The three variants V_1, V_2, and V_3 show all feasible combination among synchronization events. Note that the event s_4^{T4} will never match the events r_1^{T2} and r_2^{T2}.

Consider now that Q_0 has a new thread $T5$ containing an event send s_5, which can synchronize with r_3, r_4, and r_5. Event r_5 is a receive event inserted in thread $T3$. In this case, new race variants related to events s_3, s_4, and s_5 are present, giving in total 12 possible SYN-sequences. This example shows how reachability testing may result in a large number of SYN-sequences. Nevertheless, it is still useful because it will only generate feasible synchronization sequences, in contrast to static approaches where infeasible synchronizations must also be checked. This feature motivated the use of reachability testing to guide coverage testing, as described in Section 4.

3. STRUCTURAL TESTING FOR CONCURRENT PROGRAMS

In Souza *et al.* [12] is proposed a family of structural testing criteria for message-passing concurrent programs. To apply the testing criteria, a test model was established, which is a representation in graph form of the concurrent program designed to collect information about control, data, and communication from these programs. This model was later extended to take into account additional message-passing features, such as collective communication, non-blocking sends, distinct semantics for non-blocking receives, and persistent operations [17].

The model assumes that a fixed and known number of processes k is created at the initialization of the concurrent application. Each process may execute a different source code, and each one executes its own code in its own memory space. The concurrent program *Prog* can be specified by a set of k parallel processes: $Prog = \{p^0, p^1, \dots p^{k-1}\}$. Each process p has its own control flow graph CFG^p composed of a set of nodes N^p and a set of edges E^p [18]. Each node n in the process p is represented using the notation n^p and corresponds to a set of commands that are sequentially executed or are associated with a communication primitive (send or receive). The model supports both blocking and non-blocking receives, such that all possible interleaving between send-receive pairs can be represented.

The complete concurrent program *Prog* is thus defined by a parallel control flow graph ($PCFG$), which is composed of the individual control flow graphs CFG^p (for $p = 0 \dots k-1$) and by the representation of the communications between the processes. A synchronization edge (sync-edge) (n_i^a, n_j^b) links a *send* node in a process a to a *receive* node in a process b. Such an edge indicates the possibility of communication and synchronization between these two processes. The communication between processes supports two basic mechanisms – the first one is *point-to-point* communication (a process can send a message to another process using primitives such as *send* and *receive*), whereas the second one is *collective* communication (a process can send a message to all processes or to a particular group of them). In this model, collective communication is supported in only one predefined domain (or context) that includes all the processes in the concurrent application (using primitives that are represented in terms of basic sends). Send and receive nodes are generic representations for different types of send-receive primitives, including, for instance, blocking and unblocking primitives. These primitives are represented using the notation `send(p,k,t)` (or `receive(p,k,t)`), meaning that the process p sends (or receives) a message with tag t to (or from) the process k.

Important information about control and data flow can be derived from the $PCFG$, in particular, programs paths (both intra-process and inter-processes) together with associations of variables, which can then be used to define the coverage testing criteria. An intraprocess path is composed of a finite sequence of nodes, where all nodes are in the same process p. An interprocess path always contains at least one sync-edge pair, and this path is composed of a set of paths from different processes (including the communication edges between processes). The symbol $\prec$ is used to represent the order of execution between two nodes of different processes. Thus, $n_j^{p1} \prec n_k^{p2}$ represents that the node n_j^{p1} completes its execution before the node n_k^{p2}.

Associations of variables are triples composed of a definition node (in relation to a variable v), a node or edge (where v is used) and the variable itself (v). A variable v is defined when a value is stored in the corresponding memory position. Typical definition statements are assignments, input commands and call-by-reference function parameters. In the context of message-passing programs, a variable is also defined when it is received in a message (i.e. from a receive primitive). A use of variable v occurs when the value associated with v is referred. Three kinds of variable use are defined: *computational use (c-use)*, where the use of the variable occurs in a computation statement (a node n^p in $PCFG$); *predicate use (p-use)*, where the use of the variable occurs in a condition, or predicate, associated with control-flow statements in an intraprocesses edge (n^p, m^p) in $PCFG$; and *communication use (s-use)*, where the use of the variable occurs in a communication primitive, related to a synchronization edge (sync-edge) (n^{p1}, m^{p2}) in $PCFG$. These associations allow the detection of data-flow and communication faults during the testing of message-passing programs.

Figure 2 shows an example of a $PCFG$ for the concurrent program of Figure 3, extracted from Souza *et al.* [12]. This concurrent program is composed of four concurrent processes, where p^m is the master process and p^0, p^1 and p^2 are slave processes. The communication among the processes (sync-edge) is represented by dotted lines. For instance, the pair $(2^m, 2^0)$ is one sync-edge between processes p^m and p^0, with two communication use associations *s-use* related to the x and

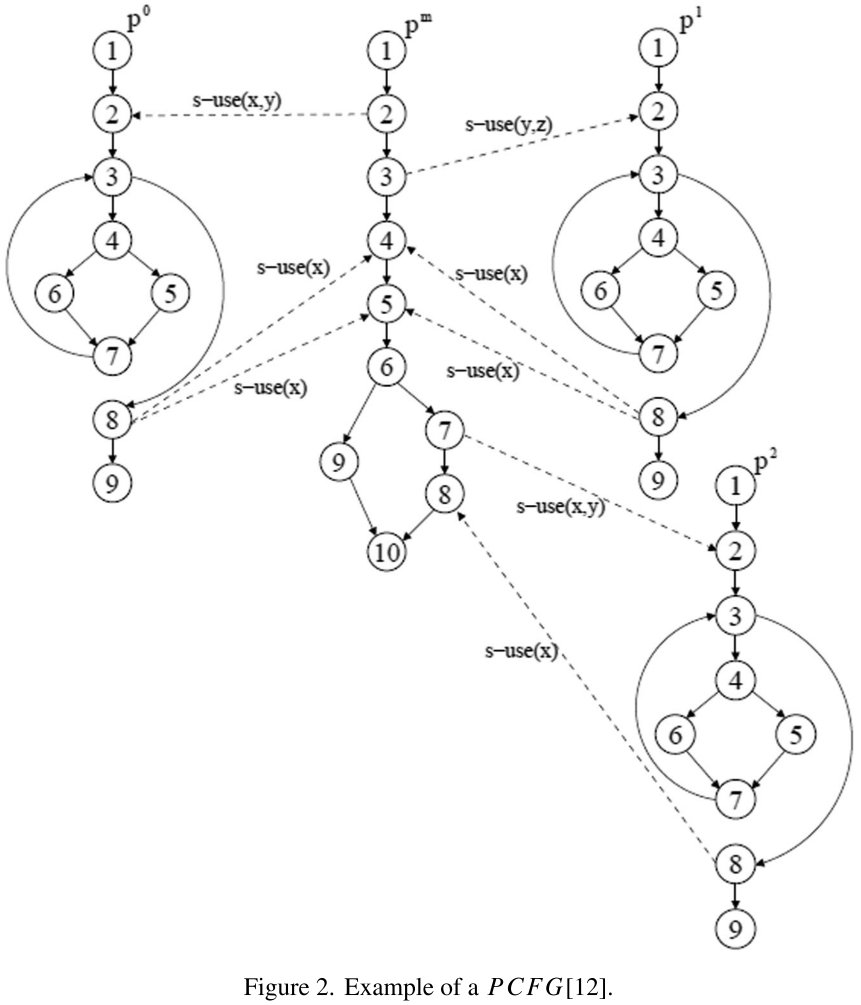

Figure 2. Example of a $PCFG$[12].

(a)

```
/* Master program GCD - mgcd.c */
#include<stdio.h>
#include "pvm3.h"
extern void pack(int);
extern int unpack();
int main(){
/*1*/ int x,y,z, S[3];
/*1*/    scanf("%d%d%d",&x,&y,&z);
/*1*/    pvm_spawn("gcd",(char**)0,0,"",3,S);
/*2*/    pack(&x);
/*2*/    pack(&y);
/*2*/    pvm_send(S[0],1);
/*3*/    pack(&y);
/*3*/    pack(&z);
/*3*/    pvm_send(S[1],1);
/*4*/    pvm_recv(-1,2);
/*4*/    x = unpack();
/*5*/    pvm_recv(-1,2);
/*5*/    y = unpack();
/*6*/    if ((x>1)&&(y>1))        {
/*7*/         pack(&x);
/*7*/         pack(&y);
/*7*/         pvm_send(S[2],1);
/*8*/         pvm_recv(-1,2);
/*8*/         z = unpack();    }
/*9*/    else { pvm_kill(S[2]);
/*9*/           z = 1;    }
/*10*/   printf("%d", z);
/*10*/   pvm_exit();      }
```

(b)

```
/* Slave program GCD - gcd.c */
#include<stdio.h>
#include"pvm3.h"
extern void pack(int);
extern int unpack();
int main(){
/*1*/    int tid,x,y;
/*1*/    tid = pvm_parent();
/*2*/    pvm_recv(tid,-1);
/*2*/    x = unpack();
/*2*/    y = unpack();
/*3*/    while (x != y){
/*4*/       if (x<y)
/*5*/            y = y-x;
/*6*/       else
/*6*/            x = x-y;
/*7*/    }
/*8*/    pack(&x);
/*8*/    pvm_send(tid,2);
/*9*/    pvm_exit();}
```

Figure 3. Greatest Common Divisor (GCD) Program in Parallell Virtual Machine (PVM): (a) master process and (b) slave process[12].

y variables. The primitives send and receive related to this sync-edge are associated with the comments /*2*/ in the master process (a) and slave process (b) (Figure 3). For simplification reasons, in the $PCFG$ are illustrated only some of the sync-edges.

Based on the test model ($PCFG$ and definitions), two sets of structural testing criteria for message-passing parallel programs are proposed, to allow testing of both sequential and parallel aspects of these programs. The first set is composed of testing criteria related to control and synchronization information:

- *a*ll-nodes criterion: this criterion requires that the test set executes paths that cover all the nodes n_i^p in $PCFG$.
- *a*ll-nodes-s criterion: this criterion requires that the test set executes paths that cover all the nodes n_i^p in $PCFG$, that contains a primitive send.
- *a*ll-nodes-r criterion: this criterion requires that the test set executes paths that cover all the nodes n_i^p in $PCFG$ that contains a receive primitive.
- *a*ll-edges criterion: this criterion requires that the test set executes paths that cover all the edges (n_j, n_k) in $PCFG$.
- *a*ll-edges-s criterion: this criterion requires that the test set executes paths that cover all the sync-edges (n_j^{p1}, n_k^{p2}) in $PCFG$.

The second set is composed of the testing criteria related to data and communication information. The main coverage criteria are the following:

- *a*ll-c-uses criterion: this criterion requires that the test set executes paths that cover all of the c-use associations.
- *a*ll-p-uses criterion: this criterion requires that the test set executes paths that cover all of the p-use associations.
- *a*ll-s-uses criterion: this criterion requires that the test set executes paths that cover all of the s-use associations.
- *a*ll-s-c-uses criterion: this criterion requires that the test set executes paths that cover all of the s-c-use associations (this association is formed by an s-use and a c-use of a variable v).
- *a*ll-s-p-uses criterion: this criterion requires that the test set executes paths that cover all the s-p-use associations (this association is formed by an s-use and a p-use of a variable v).

Table I. Required elements for PCFG example.

Criterion	Required elements
all-edges-s	$(2^m, 2^0), (3^m, 2^1), (7^m, 2^2), (8^0, 4^m), (8^0, 5^m), (8^1, 4^m), (8^1, 5^m), (8^2, 8^m)$
all-s-uses	$(1^m, (2^m, 2^0), x, y), (1^m, (3^m, 2^1), y, z), (4^m, (7^m, 2^2), x, y), (2^0, (8^0, 4^m), x), (2^0, (8^0, 5^m), x), (5^0, (8^0, 4^m), x), (5^0, (8^0, 5^m), x), (2^1, (8^1, 4^m), x), (2^1, (8^1, 5^m), x), (5^1, (8^1, 4^m), x), (5^1, (8^1, 5^m), x), (2^2, (8^2, 8^m), x), (5^2, (8^2, 8^m), x)$

$PCFG$, parallel control flow graph.

The required elements for each testing criterion are generated using the $PCFG$ to provide the minimal information that must be executed during the tests. Table I gives the required elements for the $PCFG$ example in Figure 2, for the two different testing criteria (*all-edges-s* and *all-s-uses*). In this case, the required elements are *sync-edges* and *s-uses*, respectively.

Note that for the example given in Figures 2 and 3, variables are defined in the following nodes $def(x) = \{1^m, 4^m, 2^0, 6^0, 2^1, 6^1, 2^2, 6^2\}$, $def(y) = \{1^m, 5^m, 2^0, 5^0, 2^1, 5^1, 2^2, 5^2\}$ and $def(z) = \{1^m, 8^m, 9^m\}$. This information is necessary to establish the required elements for the *all-s-uses* criterion.

Because of the inherent nondeterminism, it is impossible to determine statically whether a synchronization will be executed or not. An example is shown in the $PCFG$ given in Figure 2. In nodes 4^m and 5^m of the master process, there are nondeterministic receive primitives. In each execution, only one of the sync-edges pairs $(8^0, 4^m)$ and $(8^0, 5^m)$ will be executed, and in a similar fashion, only one of $(8^1, 4^m)$ and $(8^1, 5^m)$. Both sync-edges pairs in each example could occur during normal operation of this software and need to be executed during the testing process. This testing approach thus assumes a conservative approach, which means that all primitive sends can match all primitive receives in the program. Thus, during the static analysis all send-receive pairs are generated even though some of them might be infeasible (i.e., not executable). For instance, in Figure 2, the pair $(3^m, 2^2)$ is a theoretical send-receive pair, although in practice, this can never occur, hence, this pair is infeasible but nevertheless it is still generated during the testing activity. The advantage of this static approach is that any faults related to missing communications will always be detected.

Structural testing is a conservative approach that results in a large number of infeasible elements being tested even though they are infeasible. However, structural testing has the important advantage of using statically generated information to guide the selection of test cases and assess the coverage of the program under test. This is important because better test case selection can potentially improve the overall testing quality significantly. In contrast with reachability testing, during structural testing, it is not necessary to execute all combinations of possible synchronizations as long as at least one execution of each sync-edge pair is included, and this suffices to improve the code coverage. These advantages have motivated the proposed composite approach [14] described in the next section.

4. THE COMPOSITE APPROACH

The composite approach is a testing strategy that combines reachability testing and structural testing criteria in a systematic way. This new approach improves the testing of concurrent software by reducing the number of tests necessary for comprehensive code coverage while still maintaining the effectiveness to reveal software defects.

The main steps of the testing strategy are presented in Algorithm 1. In the algorithm, *UnCovReq* is the list of required elements not yet covered in the program under test. The *required elements* are the sync-edges and s-uses elements from the program. These testing criteria were chosen because they focus on synchronization and communication information and are therefore closely related to the information generated by reachability testing. When the testing process is complete, *UnCovReq* will be empty (assuming that complete test coverage has been achieved).

First, a list of required elements *UnCovReq* is generated from the program, using the *all-s-uses* and *all-edges-s* and the structural testing approach described in Section 3. The three following steps are carried out manually by the tester (lines 4 to 6). The tester first produces a new test case based on the program specification and the *UnCovReq* list, inserting this test case in the test set. During this process, the tester can determine that some of the required elements are in fact infeasible (i.e., they can never be executed) and these elements can be safely removed from the *UnCovReq* list without compromising the effectiveness of the testing.

The concurrent program is then executed with the test case, and an execution trace is produced containing a record of the path traversed by the test case. This path identifies which sync-edges and s-uses were executed and therefore which required elements were covered during the execution of the test case. The elements covered during the execution are removed from the *UnCovReq* list (line 8). This information is used to calculate the coverage obtained by the test set as it is constructed. In general, the complete coverage is desired, that is, all feasible required elements should be covered by the test cases in the test set.

Usually, full coverage will not be achieved at this stage, and the algorithm continues to add new test cases to the test set until 100% coverage is achieved (i.e., *UnCovReq* is empty). It is important to stress that a *required element* that is still not covered can refer to an element that may or may not be executable. If the element is executable, it can be tested by the same test case or by a new test case. Usually, the required elements that are still uncovered are executable by test cases already in the test set, but because of the nondeterminism inherent with all parallel software, they have not yet been executed. It is necessary to ensure that all possible paths and synchronizations are explored to achieve full coverage. For this reason, the next stage of the algorithm generates a list of all variants, based on the sync-edges that were executed by the test case (using the execution trace produced by line 7). Each of these variants is evaluated to determine if it has been covered or not in the testing so far, and if it has not been covered, a controlled execution is performed. This ensures that the synchronizations of each variant are always tested during program execution. After the controlled execution, the resulting execution trace is used to update *UnCovReq* (removing executed elements from *UnCovReq*). It is also possible that new variants will be identified (depending on the synchronizations) and any new variants are added to the *list_of_all_variants* (line 15).

```
   input: Program under test (PUT)
 1 test_set ⟵ ∅;
 2 UnCovReq ← static_analysis(PUT);
 3 repeat
 4     test_case ← generate_new_test_case();
 5     test_set ← add_new_test_case(test_set, test_case);
 6     UnCovReq ← mark_infeasible_elements(UnCovReq);
 7     execution_trace ← program_execution(PUT, test_case);
 8     UnCovReq ← delete_executed_elements(execution_trace, UnCovReq);
 9     if UnCovReq is not empty then
10         list_of_all_variants ← generate_variants(execution_trace);
11         for each variant ∈ list_of_all_variants do
12             for each sync-edge ∈ variant do
13                 if sync-edge ∈ UnCovReq then
14                     execution_trace ← controlled_execution(PUT, test_case, variant);
15                     list_of_all_variants ← add_new_variants(execution_trace);
16                     UnCovReq ← delete_executed_elements(execution_trace, UnCovReq);
17                 end
18             end
19         end
20     end
21 until UnCovReq is empty;
```

Algorithm 1: Composite approach algorithm.

The difference from conventional reachability testing is that only the necessary variants to execute uncovered required elements are included in the testing process. Therefore, when a new variant is selected from the *list_of_all_variants*, the first step is to confirm that it executes a new element of *UnCovReq*. If all the elements have already been tested, there is no need to continue with this variant and another variant is selected until the *list_of_all_variants* is empty. When this list is empty, it is possible that some required elements are still not covered. If this is the case, additional test cases need to be generated and the process is repeated. The algorithm concludes when *UnCovReq* is empty (i.e., all feasible elements have been covered).

To illustrate the operation of the algorithm, consider the *PCFG* in Figure 2 and the all-edges-s criterion. The list of required elements *UnCovReq* for the all-edges-s criterion is presented in Table I. Suppose that the program was executed with the test input $t = \{x = 2, y = 8, z = 4\}$ and traverses the path:

$$P^m = \{1^m, 2^m, 3^m, 4^m, 5^m, 7^m, 8^m, 10^m\}$$

$$P^0 = \{1^0, 2^0, 3^0, 4^0, 5^0, 7^0, 3^0, 4^0, 5^0, 7^0, 3^0, 4^0, 5^0, 7^0, 3^0, 8^0, 9^0\}$$

$$P^1 = \{1^1, 2^1, 3^1, 4^1, 6^1, 7^1, 3^1, 8^1, 9^1\}$$

The following sync-edges were traversed:

$$S = \{(2^m, 2^0), (3^m, 2^1), (8^0, 4^m), (8^1, 5^m), (7^m, 2^2), (8^2, 8^m)\}$$

S corresponds to the *covered required elements* for the all-edges-s criterion. These sync-edges are removed from *UnCovReq*. There are still sync-edges uncovered, and the next step is the generation of *list_of_all_variants*. The reachability testing algorithm is now executed with the test input t, and all possible variants for each sync-edge in S are generated based on the race conditions of S. Because of the *receive* nondeterministic in nodes 4^m and 5^m, there are two possible variants: $v1 = \{(2^m, 2^0), (3^m, 2^1), (8^0, 5^m)\}$, and $v2 = \{(2^m, 2^0), (3^m, 2^1), (8^1, 4^m)\}$. Applying the reachability testing algorithm, each *variant* is used to conduct a prefix-based test, forcing the sync-edges in those variant to be replayed and then allowing the test run to proceed nondeterministically. In other words, each variant contains only the sync-edges that will be executed in a prefix-based test run. Each variant is able to cover one or more of the uncovered sync-edges in *UnCovReq*. For example, $(8^0, 5^m)$ and $(8^1, 4^m)$ are two required elements in Table I that have not yet been executed. A controlled execution of variant $v1$ is performed, and the traversed path is collected to analyse which required elements in *UnCovReq* are covered. In this new execution, the same paths are traversed; however, the synchronization order is $S = \{(2^m, 2^0), (3^m, 2^1), (8^0, 5^m), (8^1, 4^m), (7^m, 2^2), (8^2, 8^m)\}$. This execution is able to cover the required elements of the all-edges-s criterion not yet covered, achieving the coverage goal for this testing criterion.

The main advantage of this composite strategy presented earlier is to reduce the computational cost of testing. It can either be applied to improve structural testing (by improving the coverage analysis) or alternatively to improve reachability testing (by guiding the selection of the SYN-sequences). As explained earlier, the exhaustive testing necessary with reachability testing to execute all SYN-sequences is not always practical, and empirical mechanisms are necessary to guide the selection of appropriate SYN-sequences. The composite approach presented in this section provides an efficient mechanism to select the SYN-sequences required to cover the additional elements necessary for a given testing criterion.

5. EXPERIMENTAL STUDY

This section reports the definition, design, execution, and analysis of an experimental study, following the experimental software engineering process, defined by Wohlin [15]. The *goal* of this study is to evaluate the application cost and effectiveness of the composite approach, compared with reachability testing and structural testing. The *quality focus* is related to application cost and effectiveness. The *context* of the experiment refers to concurrent programs implemented in C using message-passing programs in MPI.

Table II. Characteristics of the programs used in the experiment.

Programs	Processes	LOC	Sends	Receives
sieve	4	114	7	7
gcd	4	111	7	5
mmult	4	198	15	27
philosophers	10	152	29	28
pairwise	8	176	32	32
reduction	6	133	6	6
qsort	4	480	17	29
jacobi	6	525	71	109
Van der Walls	4	552	4	4

LOC, lines of code

5.1. Experiment Planning

The ValiMPI testing tool was used to conduct this study – ValiMPI is a tool developed to test concurrent programs implemented using MPI [19], proposed originally to support the coverage testing described in Section 3. The tool provides functionalities to (i) create test sessions; (ii) save and execute test data; and (iii) evaluate the testing coverage with respect to a given testing criterion. ValiMPI has been extended to implement the reachability testing strategy proposed by Lei and Carver [9], and it also supports the composite approach presented in this paper.

Nine different MPI programs were used in this study, eight of them implementing different classical concurrency algorithms while one is a concurrent applications program from the bioinformatics domain [20]. These programs contain typical synchronization/communication patterns that are commonly found in many practical applications, including numerical problems, sorting algorithm, producer-consumer, and bioinformatics.

Table II shows the complexity of the programs in terms of number of processes lines of code and send/receive primitives. It is worth emphasizing that with concurrent software, the complexity of a program depends more on the way in which the processes communicate and synchronize rather than the size of the program.

1. *sieve of Eratosthenes* - to find all prime numbers up to a specified integer [21];
2. *gcd* to calculate the greatest common divisor of three numbers, using successive subtractions between two numbers until one of them is zero;
3. *mmult* to implement matrix multiplication using data/domain decomposition;
4. *philosophers* to implement the classical dining philosophers problem;
5. *pairwise* to calculate interprocess interactions. Each process n_i receives data X_i and is responsible for computing the interactions $I(X_i, X_j)$, for $i \neq j$. For this, a structure with N channels is used, where each communication channel represents a source-destination pair – these channels are used to connect the N tasks into a unidirectional ring;
6. *reduction* to implement the reduction operation of distributed data considering add, multiplication, greater than and less than operations;
7. *qsort* to implement quicksort based on the parallel algorithm presented in Grama [22];
8. *jacobi* to implement the Jacobi-Richardson iterative solution of a linear system of equations; and
9. *Van der Waals* to calculate the Van der Waals energy of a protein using a genetic algorithm [20].

Based on the objectives of the experimental study, two research questions and hypotheses were defined and used to analyze the results. *CovT* refers to the structural testing criteria, RT refers to reachability testing, and *RTCovT* refers to the composite approach.

- *Research question Q1*: *Is RTCovT able to reduce the computational cost of concurrent program testing?*
  Null hypothesis 1 (NH1): the cost is the same for *RTCovT* and RT.
  NH1: cost (*RTCovT*) = cost(RT).

- *Alternative hypothesis 1.1 (AH11)*: the cost of *RTCovT* is greater than the cost of RT. AH11: cost (*RTCovT*) $>$ cost(RT).
- *Alternative hypothesis 1.2 (AH12)*: the cost of *RTCovT* is less than the cost of RT. AH12: cost (*RTCovT*) $<$ cost(RT).

Null hypothesis 2 (NH2): the cost is the same for *RTCovT* and *CovT*.
NH2: cost (*RTCovT*) $=$ cost(*CovT*).

- *Alternative hypothesis 2.1 (AH21)*: the cost of *RTCovT* is greater than the cost of *CovT*. AH21: cost (*RTCovT*) $>$ cost(*CovT*).
- *Alternative hypothesis 2.2 (AH22)*: the cost of *RTCovT* is less than the cost of *CovT*. AH22: cost (*RTCovT*) $<$ cost(*CovT*).

• *Research question Q2*: *What is the effectiveness of RTCovT for concurrent programs testing?*
Null hypothesis 3 (NH3): the effectiveness is the same for *RTCovT* and *CovT*.
NH3: effectiveness (*RTCovT*) $=$ effectiveness(*CovT*).

- *Alternative hypothesis 3.1 AH31)*: the effectiveness of *RTCovT* is greater than the effectiveness of *CovT*. AH31: effectiveness (*RTCovT*) $>$ effectiveness(*CovT*).
- *Alternative hypothesis 3.2 (AH32)*: the effectiveness of *RTCovT* is less than the effectiveness of *CovT*. AH32: effectiveness (*RTCovT*) $<$ effectiveness(*CovT*).

The design of the experiment is composed of the following steps:

1. *Generation of a test set $T1$, using the structural testing criteria (CovT)*: a test set $T1$ was manually generated based on *all-s-uses* and *all-edges-s* criteria. Test cases in $T1$ execute all feasible required elements of these criteria (s-use associations and sync-edges, respectively). In this step, it was necessary to identify the infeasible elements in order to define the coverage of $T1$. Thus, $T1$ is a test set adequate to the *all-s-uses* and *all-edges-s* criteria, because $T1$ is able to cover the required elements of these criteria.
2. *Execution of reachability testing (RT)*: reachability testing was executed with $T1$ (generated by *CovT*), using the original algorithm proposed by Lei and Carver [9].
3. *Execution of the composite approach (RTCovT)*: based on the *all-s-uses* and *all-edges-s* structural criteria, the composite approach was executed following the steps discussed in Section 4. For this step, two test sets were employed: $T1$ and $T2$, the latter created by the tester using the *RTCovT* approach.
4. *Evaluation of the effectiveness to detect faults*: in this step, the ability to reveal faults using *RTCovT* was evaluated. For this, defects were inserted in each program and these programs were executed with $T1$ and $T2$ and the results assessed. To compare the effectiveness, a randomly-generated test set was employed to execute the faulty programs. Details of these defects are presented in Section 5.2.4.

Before the experiment execution, the study environment was prepared. The programs were selected, a set of defects to be considered was specified, the method of defect insertion was defined, and the ValiMPI testing tool was installed. The experiment environment has the following features: GNU/Linux operational system using the Ubuntu 11.10 distribution with 3.0.0-32-generic-pae, 4.1.2 gcc compiler, Open MPI 1.4 and the ValiMPI testing tool [19].

5.2. Execution of the Experiment

5.2.1. Execution of the structural testing – CovT. Initially, the test sets $T1$ were generated for the programs where each $T1$ traverses all feasible required elements of both *all-s-uses* and *all-edges-s* testing criteria. The programs were executed with the test set using the ValiMPI tool and observing the coverage obtained. New test cases were added until all feasible required elements were executed and the maximal coverage was obtained. In this phase, the infeasible elements were manually identified. In order to address the nondeterminism, the following approach was adopted. Initially, $T1$ was repeatedly executed in order to execute different synchronizations. In fact, these repetitions do not guarantee that any new synchronization will occur, but it is a relatively inexpensive approach.

Table III. Results from the application of the *CovT* approach.

	Size	All-edges-S		All-S-Uses	
Programs	of $T1$	Sync-edges	Infeasible	S-uses assoc.	Infeasible
sieve	4	39	18	57	36
gcd	7	14	4	34	15
mmult	4	189	144	252	202
philosophers	-	81	27	108	36
pairwise	3	896	864	1344	1304
reduction	4	30	25	60	55
qsort	6	171	78	300	190
jacobi	6	151	32	255	130
Van der Walls	2	66	54	156	143

Table IV. Number of the SYN-sequences executed by *CovT*, RT and *RTCovT* approaches.

Programs	Size of	$T1$			Size of	$T2$
	$T1$	CovT	RT	RTCovT	$T2$	RTCovT
sieve	4	840	144	84	2	41
gcd	7	700	126	70	4	40
mmult	4	1680	432	168	3	126
philosophers	—	450	45	45	—	45
pairwise	3	240	24	24	3	24
reduction	4	200	20	20	4	20
qsort	6	1560	1404	156	11	286
jacobi	6	1860	1674	186	8	248
Van der Walls	2	440	44	44	2	44

In remaining cases where synchronizations had not been executed, the controlled execution was employed, forcing the execution of these synchronizations. Controlled execution mechanism follows the same ideas proposed by Carver and Tai [10].

Table III shows the results from the application of the *CovT* approach, giving the size of $T1$ (test set adequate to the criteria) and the number of required elements for each testing criterion adopted in the study. The sync-edges column contains the total of required elements generated by the all-edges-s criterion, including infeasible elements (the total of infeasible elements is also shown for each criterion). Equivalent information is shown for the all-s-uses criterion. The number of SYN-sequences executed to cover these elements is presented in Table IV and discussed in Section 5.2.3.

5.2.2. Execution of the reachability testing approach – RT. Reachability testing was applied using the test set $T1$ generated previously. For this, one SYN-sequence for each test case in $T1$ was collected and from each SYN-sequence, the race variants were generated, producing new SYN-sequences. In this case, it does not matter whether the synchronization has already been traversed in some previous execution; it is executed again to guarantee the execution of all possible synchronization combinations. Table IV gives the number of SYN-sequences executed by RT (discussed in the next section).

5.2.3. Execution of the composite testing approach – RTCovT. In this phase, the composite approach algorithm was executed, and two different scenarios were analysed. The first one is the SYN-sequences executed by RT, *RTCovT* and *CovT* using the same test set $T1$. The difference is that in the *RTCovT* approach, there is a systematic method to select what synchronizations will be executed, reducing the number of SYN-sequences executed compared with RT and *CovT*.

In the second analysis, a new test set $T2$ was used, simulating a real test scenario where the tester is using *RTCovT* to conduct the testing activity. $T2$ was generated using the *RTCovT* approach (based on the *all-s-uses* and *all-edges-s* testing criteria).

Table V. Evolution of the test coverage for the *jacobi* program.

	All-Edges-S		All-S-Uses	
testcases	CovT	RTCovT	CovT	RTCovT
$tc1$	19.3%	19.3%	8.7%	8.7%
$tc2$	38.6%	49.1%	26.1%	34.8%
$tc3$	82.5%	94.7%	60.9%	71%
$tc4$	84.2%	96.5%	68.1%	78.3%
$tc5$	94.7%	100%	78.3%	82.6%
$tc6$	100%	100%	78.3%	82.6%

Table VI. Evolution of the test coverage for the *mmult* program.

	All-Edges-S		All-S-Uses	
testcases	CovT	RTCovT	CovT	RTCovT
$tc1$	60%	93.3%	63.2%	89.5%
$tc2$	60%	93.3%	68.4%	94.7%
$tc3$	73.3%	93.3%	78.9%	94.7%
$tc4$	86.7%	93.3%	89.5%	94.7%

Table IV shows the number of SYN-sequences executed by RT, *RTCovT* and *CovT* using $T1$ and also for *RTCovT* using $T2$.

For all the programs, the number of SYN-sequence executed by *RTCovT* was less when compared with the *CovT* approach and, for most of the programs, it was also less when compared with the RT approach. Because of the particular features of the programs *philosophers, pairwise, reduction* and *Van der Walls*, the number of SYN-sequence executed by RT and *RTCovT* were equal. These results demonstrate that computational cost can be reduced using the *RTCovT* approach. A statistical analysis is presented in Section 5.3.

Table V provides additional details for the *jacobi* program and illustrates the different coverage of the *CovT* and *RTCovT* approaches. For each test case generated ($tc1$ to $tc6$), it is presented the coverage obtained for both testing criteria adopted in the study. The results indicate that *RTCovT* always provides a better test coverage, moreover, the maximal coverage is achieved after five test cases have been added, meaning that these five test cases are enough to obtain maximal coverage. Similar results are shown in Table VI for the *mmult* program. In this case, the maximum coverage is achieved after just two of the test cases have been added. Note that the desired maximum coverage is always 100%; however, in some cases, this is not possible because of the infeasible elements.

5.2.4. Evaluation of fault detection effectiveness. In order to evaluate the effectiveness of the composite approach, the ability in detecting defects introduced into the programs was analysed. Defects were injected based on the existent classification of defects for concurrent programs [8, 23, 24]. A strategy similar to mutation testing [25] was applied, where defects were systematically inserted in each program, generating a new program for each inserted defect. According to Andrews *et al.* [26], mutation is an appropriate approach for testing experiments because it avoids introducing any bias. Table VII shows the type of defects considered and the number of defects for each program, totaling 374 faulty programs.

Each faulty program was executed with the test sets $T1$ (for the *CovT* approach) and $T2$ (for the *RTCovT* approach) and their ability to reveal the defects was determined. Also, an additional test set (TS_{rand}) was generated, containing 30 test cases randomly generated according to the input domain of each program. The effectiveness of the TS_{rand} was evaluated in comparison with the *RTCovT* approach. Table VIII shows the percentage of defects revealed by *CovT*, *RTCovT* and TS_{rand}, representing the effectiveness of each approach as defined by the following equation:

Table VII. Number of defects inserted in each program.

Type of defect	gcd	jacobi	mmult	qsort	red.	sieve	philos.	pair.	Waals
Incorrect loop or selection structure	2	8	0	2	3	1	1	3	1
Incorrect process in messages	2	4	1	0	2	1	1	4	0
Source process changed by “any” process in messages	1	5	0	2	0	1	0	4	0
Incorrect size of array	2	4	3	1	0	4	0	2	0
Non initialized variable	0	5	4	0	3	1	5	5	2
Incorrect size of message	0	2	4	1	0	2	2	0	2
Incorrect message address	2	4	2	4	4	6	2	3	4
Incorrect type of parameter	1	3	3	1	1	0	1	3	0
Incorrect message data type	1	1	1	1	0	4		3	2
Replacement blocking by non-blocking message	5	1	1	2	2	4	0	4	2
Changing of operator in attributions	2	8	5	3	2	8	3	4	5
Incorrect data sent or received	1	5	5	3	2	5	3	1	4
Change the logical operator in predicative statements	2	5	3	5	1	7	3	3	4
Missing statements	5	8	6	4	2	7	1	4	5
Incorrect attributions	4	4	4	5	1	7	0	3	5
Increment/decrement of variables in communication primitives	0	4	2	1	0	4	0	4	0
Total	**30**	**71**	**44**	**35**	**22**	**60**	**23**	**50**	**39**

Table VIII. Effectiveness of the *RTCovT* approach in comparison to the *CovT* and the random test set.

Programs	Number of defects	$CovT(T1)$	$RTCovT(T2)$	TS_{rand}
sieve	60	98.3%	100%	100%
gcd	30	100%	100%	96.7%
mmult	44	90.9%	97.7%	97.7%
pairwise	50	100%	100%	94%
reduction	22	100%	100%	98%
qsort	35	100%	100%	100%
jacobi	71	94.4%	100%	36.6%
Van der Waals	39	100%	100%	89.7%

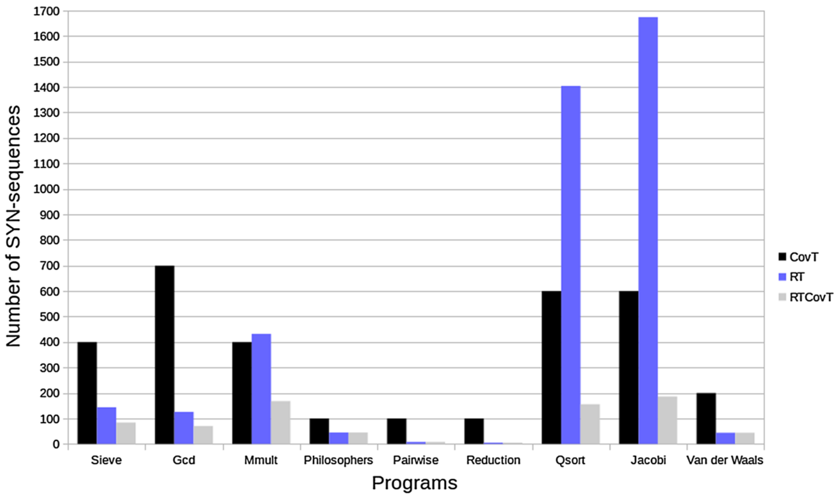

Figure 4. Computational cost of the *RTCovT*, RT and *CovT* testing approaches.

$$effectiveness = \frac{number\ of\ faults\ found}{number\ of\ faults\ injected} \tag{1}$$

The results indicate that the *RTCovT* approach is highly effective in detecting defects and is capable of finding most of the injected defects. For the *sieve* and *jacobi*, programs the defects were revealed only by the *RTCovT* approach.

For the *mmult* program, nine defects were not detected by the *CovT* approach but only one using the RTCovT approach. Note that the *RTCovT* approach was always able to detect more defects than the randomly test sets, indicating that the selection of test cases guided by coverage testing criteria provides improved results.

5.3. Result Analysis

The analysis and interpretation of the results are made based on descriptive statistics and hypothesis testing. Parametric and non-parametric tests [15] were used to determine whether it is possible to reject the null hypothesis based on collected data set and statistical tests. The descriptive analysis is useful to describe and to display graphically interesting aspects of the study.

There are several different perspectives that could be used to evaluate the cost of a testing criterion. In this study, it was chosen to use the number of SYN-sequences (or paths) executed for each approach. Figure 4 presents the amount of SYN-sequences traversed by each approach using the test set $T1$. These results suggest that the *RTCovT* approach provides a lowest cost compared with the *CovT* and RT approaches. The *CovT* approach has a higher cost because the tester usually needs to execute the program several times to deal with the nondeterminism.

To analyse the null hypothesis $NH1$ (*the cost is the same for RTCovT and* RT), the *Shapiro-Wilk test* was applied to evaluate whether the population is normally distributed. The obtained p-value for

Table IX. Results of the Wilcoxon test for the statistics analysis of cost of the *RTCovT* and RT approaches.

Data	Alternative	W	p-value
cost of *RTCovT* and cost of RT	two.sided	34	0.5652
cost of *RTCovT* and cost of RT	less	34	0.2826
cost of *RTCovT* and cost of RT	greater	34	0.7174

Table X. Results of the t-test analysis for the statistics analysis of cost of the *RTCovT* and *CovT* approaches.

Data	Alternative	t	df	p-value
cost of *RTCovT* and cost of *CovT*	two.sided	−3.6782	8.16	0.006025
cost of *RTCovT* and cost of *CovT*	less	−3.6782	8.16	0.003012
cost of *RTCovT* and cost of *CovT*	greater	−3.6782	8.16	0.0997

the *RTCovT* sample is 0.1133 and it is greater than 0.05, meaning this sample is normal. For the RT sample, the p-value is 0.001043 and less than 0.05, meaning this sample is not normal. Therefore, these results suggest the application of a non-parametric test, in case, the *Wilcoxon rank sum test*. Thus, the *Wilcoxon test* was applied using the samples for *RTCovT* and RT and the obtained *p-value* was 0.5652 (Table IX). This value is greater than 0.05 hence $NH1$ was accepted with confidence interval of 95%. This result indicates that there are no significant difference in the costs of these two testing approaches, although the descriptive statistics suggest that *RTCovT* has a lower cost when compared with RT (Figure 4).

In relation to the null hypothesis $NH2$ (*the cost is the same for RTCovT and CovT*), the *Shapiro-Wilk test* was applied to verify the normality of the data. As the two samples are normal, the F-test was applied to compare the variances of the samples. The results of the F-test indicate that the values of the two samples are not equal, and the parametric t-test can be applied. Table X presents the results obtained for t-test analysis, considering the three alternatives to be analysed. For this test, the *two-sided alternative* was initially considered in which the obtained *p-value* was 0.006025, which is less than 0.05 indicating that there is a cost difference between *RTCovT* and *CovT* and, consequently, $NH2$ can be rejected.

In next step, the parametric t-test was applied to evaluate alternative hypothesis $AH21$ and $AH22$, using the *less alternative*. The objective is evaluating if the cost of *RTCovT* is less than the cost of *CovT*. According to Table X, the obtained p-value was 0.003012, which is less than 0.05, meaning that the cost of *RTCovT* is less than the cost of *CovT*. To confirm this result, the t-test was again applied using the *greater alternative*, in which the p-value obtained was 0.0997, greater than 0.05 confirming that the cost of *RTCovT* is less than the cost of *CovT* and, therefore, the alternative hypothesis $AH22$ is accepted with a confidence level of 95%.

Figure 5 shows the effectiveness in terms of the quantity of defects revealed for *RTCovT* and *CovT* in relation to the total of injected defects. Results indicate that *RTCovT* maintains good effectiveness even though it traverses a lesser number of SYN-sequences.

Null hypothesis $NH3$ (*the effectiveness is the same for RTCovT and Covt*) was analysed by comparing the results for the effectiveness of test sets $T1$ and $T2$. The nonparametric *Wilcoxon test* was applied and the p-value obtained for the pair *CovT*($T1$) and *RTCovT*($T2$) was 0.2697 (Table XI). This result indicates that there is no significant difference between the effectiveness of these two testing approaches and, therefore, $NH3$ is accepted. Results from the descriptive statistics and from the hypothesis testing suggest that the *RTCovT* approach has significant potential to support concurrent program testing more efficiently, taking into account both the computational cost and the effectiveness to reveal faults.

The effectiveness of *RTCovT* and TS_{rand} was also statistically analyzed. For this, the *Wilcoxon test* was applied and the results are presented in Table XII. The p-value obtained was 0.00749, indicating that *RTCovT* presents higher effectiveness than TS_{rand}.

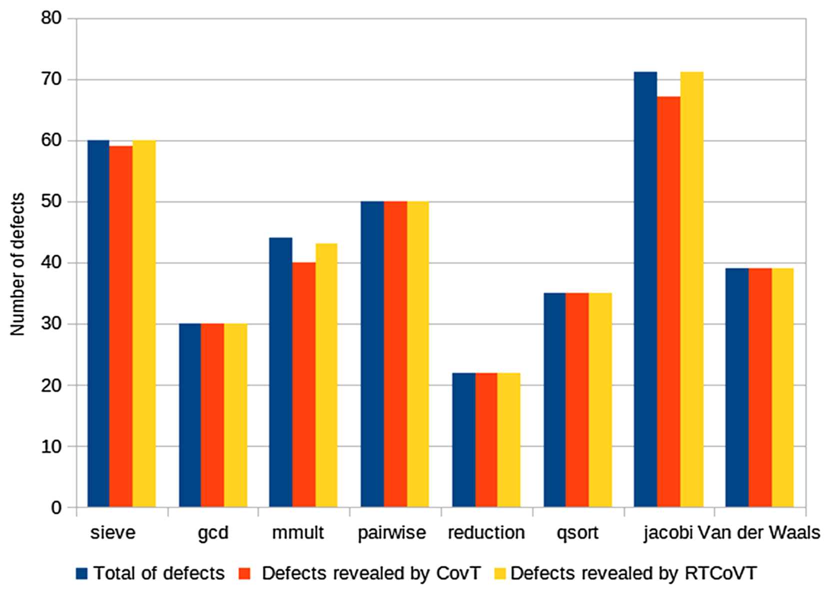

Figure 5. Effectiveness of the *CovT* and *RTCovT* approaches.

Table XI. Results of the Wilcoxon test for the statistics analysis of effectiveness of the *RTCovT* approach.

Data	W	p-value
effectiveness of *RTCovT* and *CovT*	40.5	0.2697

Table XII. Results of the t-test analysis for the statistics analysis of effectiveness of the *RTCovT* and *CovT* approaches.

Data	alternative	W	p-value
cost of *RTCovT* and cost of *CovT*	two.sided	53.5	0.01500
cost of *RTCovT* and cost of *CovT*	less	53.5	0.99460
cost of *RTCovT* and cost of *CovT*	greater	53.5	0.00749

Results from the hypothesis testing suggest that the *RTCovT* approach has a significant potential to improve the testing effectiveness, taking into account both the computational cost and the effectiveness to reveal faults when compared to the random approach.

5.4. Threats to Validity

In the design of the experimental study, it is important to identify possible threats to the validity of the results and mechanisms to mitigate these threats need to be considered. Based on the classification proposed by Wohlin [15], the following threats which could affect the validity of the experimental study were identified:

Internal validity: this type of threat is concerned with the question of whether the inputs to the experimental study actually caused the results obtained. For example, the defects injected into the example programs, could have be inserted based on the knowledge and understanding of the researcher about the testing criteria being used. This potential threat was addressed by adopting a strategy similar to mutation testing to insert the defects automatically without potential researcher bias.

Conclusion validity: this type of threat can affect the ability to draw conclusions about relationship between the treatment and the results obtained. The test case sets need to be designed according to a specified coverage criteria. It is known that the ability to reveal faults is related to a particular test case, and it is important to evaluate if the adopted criterion contributes to the selection of effective test cases or not. To mitigate this threat, the test sets were compared against random test cases, to observe what the influence of coverage testing might be in selecting effective test cases. The results indicate that there is an appropriate relationship between effectiveness and the selection of test cases guided by coverage criteria.

External validity: this is concerned with whether it is possible to generalize the results of the study. In this study, this threat is related to the selected programs used in the experimental study, in which two aspects were considered during the selection of the programs: complexity and concurrency. To mitigate this threat, the selected programs present a range of different numbers of sending and receive statements, with different sizes and number of processes. These programs are widely used in other studies and contain classical communications mechanisms found in many practical concurrent programs. This study primarily considered the concurrency-related complexity of the programs but other complexities, such as control flow, data declaration, data flow and arithmetic metrics, may also interfere in the computational cost of the proposed approach. These aspects were not explored in this study and will be considered in further investigations.

6. RELATED WORK

Published research in the general area of concurrent program testing has covered aspects such as failure injection [27], static analysis [28–30], controlled execution [31–37], mutation testing [38–40], model checking [41–44], model-based testing [45, 46], coverage testing [47–60], symbolic analysis [61, 62], reachability testing [9, 16, 63–72] and test case generation [73–75]. In this section are discussed only the relevant related research. Souza *et al.* [76] presents a systematic review, with a larger set of references.

Taylor *et al.* [1] propose a set of structural coverage criteria for concurrent programs based on the notion of concurrent states and concurrency graph. Five criteria are defined: *all-concurrency-paths*, *all-proper-cc-histories*, *all-edges-between-cc-states*, *all-cc-states*, and *all-possible-rendezvous*, with an analysis of the hierarchy between these criteria. The authors stress that every approach based on reachability analysis will be limited by state space explosion.

Chung *et al.* [77] propose four different testing criteria for Ada programs: *all-entry-call*, *all-possible-entry-acceptance*, *all-entry-call-permutation*, and *all-entry-call-dependency-permutation* (these criteria focus on the rendezvous between tasks). They also present a hierarchy based on these criteria.

Yang *et al.* [78, 79] extended the data flow criteria to shared-memory parallel programs. A *parallel program flow graph* is constructed and traversed to obtain the paths, variable definitions, and uses. The test requirements are all paths that have definition and use of variables related to thread parallelism. The *Della Pasta Tool (Delaware Parallel Software Testing Aid)* automates their approach. The authors present the foundations and theoretical results for the structural testing of parallel programs, with a definition of the *all-du-path* and *all-uses* criteria for shared-memory programs. Their study inspired the test model definition for message-passing parallel programs, which was described in Section 3.

Wong *et al.* [3] propose a set of methods to generate test sequences for the structural testing of concurrent programs. The reachability graph is used to represent the concurrent program and to select test sequences for the *all-node* and *all-edge* criteria. The methods are designed to generate a small set of test sequences to cover all nodes and edges in a reachability graph, providing information about the parts of the program to be covered to increase the effective coverage of these criteria. The authors stress that the major advantage of the reachability graph is that only feasible paths will be generated. However, they do not explain how to generate the reachability graph from the concurrent program or how to deal with the state space explosion (which is inevitable with larger problems).

Kojima *et al.* [54] employ the *all-concurrent-paths* criterion on a Concurrent Module Flow Graph to execute the structural testing of embedded concurrent software present in different devices, such as high-definition TVs, recorders and mobile phones. To overcome the very large quantity of test cases required when the number of elements increases, the authors propose to suppress test cases by considering the *happens-before* relation between local blocks (those that do not include operating shared resources) and focus the testing activity on the external operation blocks instead (those operating on shared resources). This approach does not directly address the large quantity of synchronization pairs that are derived in the static analysis.

Hong *et al.* [72] introduced a technique to achieve high coverage for concurrent program entities, such as statement pairs and synchronization pairs. The technique consists of an estimate phase and a testing phase. In the estimate phase, the technique dynamically builds a thread model of the target concurrent program, and, based on the model, estimates the coverage requirements. In the testing phase, the technique dynamically manipulates thread schedules to cover previously uncovered coverage requirements. The authors provide results of an empirical study, indicating that the technique is more effective and efficient in achieving high synchronization coverage than the random testing technique for a suite of Java programs.

Tasiran *et al.* [80] present a coverage metric, called location pairs, developed to assist in the testing of shared-memory concurrent programs. Location pairs is formed by points where a shared variable is used in the threads. The objective is to cover access patterns likely to lead to a particular type of concurrency error. According to the authors, this coverage metric can evaluate how thoroughly a program has been tested and guide testing toward unexplored concurrency scenarios.

Edelstein *et al.* [7, 81] present a multi-threaded bug detection architecture called *ConTest* for Java programs. This architecture combines a replay algorithm with a seeding technique, in which the coverage is specific to race conditions. The seeding technique seeds the program with *sleep* statements at shared memory access, and synchronization events and heuristics are used to decide when a *sleep* statement must be activated. These heuristics are defined based on the bug patterns for concurrent programs [8]. The replay algorithm is used to re-execute a test when race conditions are detected, ensuring that all accesses in race will be executed. The focus of the work is the nondeterminism problem, rather than code coverage and testing criteria.

Yu *et al.* [82] propose a coverage-driven testing tool called Maple, which seeks the exploration of untested interleaved thread. The authors define coverage for multi-threaded programs based on a set of thread-interleaving idioms, which represents the common cases of shared variable interleaving. The approach avoids testing the same thread interleaving across different test inputs. The results indicate that the approach can detect faults faster by exposing more untested interleaving in a shorter period of time than using conventional methods.

Reachability testing was proposed by Hwang *et al.* [16] and has been widely investigated because of its advantages and the limitations experienced in practice. Reachability testing is an approach for the dynamic testing of concurrent program that executes different synchronization sequences without constructing any static model. Tai [63] describes the application of the reachability testing to concurrent programs using message passing. Lei and Carver [67] provide details of how to apply reachability testing to semaphore-based programs. Li *et al.* [68] provide a framework to use reachability testing in the context of Java multi-thread programs. In contrast to previous work, in this research, the selection of synchronization sequences is based on an analysis of reading and writing shared variables. Gong *et al.* [83] extend the reachability testing for Java programs, considering events among multiple synchronization objects.

Lei and Carver [9] present a general execution model that allows reachability testing to be applied to several commonly used synchronization constructs. They also present a new method for performing reachability testing, which guarantees that every partially ordered synchronization sequence will be traversed exactly once without saving any sequences already exercised.

Lei *et al.* [64] propose a combinatorial testing strategy, called t-way reachability testing, which selectively exercises race variants for each synchronization sequence. The objective is to reduce the cost by selecting a subset of synchronization sequences to be exercised. The authors provide results that indicate that the strategy can select effective synchronization sequences.

Carver and Lei [69] describe an approach to test monitors in multi-threaded programs using reachability testing to derive all possible orders in which testing method calls can enter in the monitor. The authors also show how to store and recognize visited states when reachability testing is applied to a single monitor.

Carver and Lei [66] propose a distributed reachability testing algorithm, allowing different test sequences to be executed concurrently. This algorithm reduces the time to execute the synchronizations, but the authors do not comment on the effort necessary to analyse the results from these executions.

Hwang *et al.* [84] propose an approach that employs reachability testing to achieve statement coverage in the dynamic testing of concurrent programs. In this approach, test inputs are derived based on SYN-sequences obtained during the reachability testing and used to perform reachability testing multiple times to achieve statement-coverage testing for a concurrent program. However, the authors did not use information about coverage to execute new statements; they derive test inputs from a previous execution of reachability testing. This approach is similar to our work, in that the focus is executing additional statements in order to improve the code coverage.

Ratsaby *et al.* [85] introduce the concept of coverability, which refers to the degree to which a state-machine model can be covered when subjected to testing. Coverability analysis establishes goals and during the test is observed if these goals are reached or not. The authors use model checker to improve the coverability, defining temporal rules to check some properties, for instance, the code reachability. Based on static program analysis and on simulation of counter examples, the authors define heuristics that can be combined to improve the coverability analysis of the concurrent programs. This paper use some concepts and objectives slightly similar to our work; however, the authors focus on properties verification and the model checker while this paper is concerned with code coverage using structural testing criteria to measure the coverage.

7. CONCLUSIONS

This paper has presented the results of an experimental study conducted to evaluate the composite approach, designed to test concurrent software by combining both structural testing and reachability testing. In the composite approach, the structural testing criteria are used to select test cases and determine the synchronizations to be executed, always selecting new synchronizations whenever possible. Reachability testing is employed to provide a set of synchronizations to be traversed.

The experimental study was conducted to evaluate two factors: testing cost and fault detection effectiveness. With respect to testing cost, the composite approach was compared with both structural testing and reachability testing, basing the comparisons on the number of synchronizations executed. The results demonstrate that the composite approach presents a low computational cost compared with structural testing, executing less synchronizations to achieve the maximum coverage for the structural testing criteria. Also, the composite approach requires fewer test cases to obtain the maximum coverage.

To evaluate the effectiveness, defects were systematically inserted into the programs, and the ability to detect them was evaluated by comparing the test sets generated for structural testing, composite approach, and random testing. The results demonstrate that the composite approach is able to select test sets with high effectiveness, detecting the most of the defects seeded into the programs.

The results indicate that the composite approach can improve the quality of tests, reducing the computational cost and also maintain the effectiveness of detecting faults. In this paper, the composite approach is used to guide the selection of test cases based on structural testing criteria and applying reachability testing to improve the test coverage. An alternative is to use the composite approach to minimize the number of sequences executed in the reachability testing; in this case, the coverage criteria are used to indicate which race variants should be chosen in each execution.

As part of future work, an investigation into the mapping other structural testing criteria onto the composite approach will be undertaken. Furthermore, this approach will be implemented in the context of shared memory programs, using the testing criteria defined in Sarmanho *et al.* [13]. New experimental studies will be defined for the evaluation of the composite approach, for instance, using different test sets for the reachability testing and the composite approach, in order to evaluate the effectiveness of the both approaches.

ACKNOWLEDGEMENTS

This work is sponsored by FAPESP and CAPES 2008/04614-5, 2009/04517-2, 2010/02839-0, 2013/01818-7, 1191/10-1.

REFERENCES

1. Taylor RN, Levine DL, Kelly C. Structural testing of concurrent programs. *IEEE Transaction on Software Engineering* 1992; **18**(3):206–215.
2. Yang CSD. Program-based, structural testing of shared memory parallel programs. *Ph.D. Thesis*, University of Delaware, 1999.
3. Wong WE, Lei Y, Ma X. Effective generation of test sequences for structural testing of concurrent programs. *10th IEEE International Conference on Engineering of Complex Systems (ICECCS 2005),* Shanghai, China, 2005; 539–548.
4. Lu S, Jiang W, Zhou Y. A study of interleaving coverage criteria. *Proceedings of the ACM SIGSOFT symposium on the foundations of software engineering*, ACM: New York, NY, USA, 2007; 533–536.
5. Robinson-Mallett C, Hierons RM, Poore J, Liggesmeyer P. Using communication coverage criteria and partial model generation to assist software integration testing. *Software Quality Control* 2008; **16**(2):185–211.
6. Takahashi J, Kojima H, Furukawa Z. Coverage based testing for concurrent software. *28th International Conference on Distributed Computing Systems Workshops,* Beijing, China, 2008; 533–538.
7. Edelstein O, Farchi E, Goldin E, Nir Y, Ratsaby G, Ur S. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience* 2003; **15**(3-5):485–499.
8. Farchi E, Nir Y, Ur S. Concurrent bug patterns and how to test them. *17th International Parallel and Distributed Processing Symposium (IPDPS 2003) - Workshop on Parallel and Distributed Systems: Testing and debugging*, IEEE Computer Society: Nice, France, April 2003; 286–293.
9. Lei Y, Carver RH. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering* 2006; **32**(6):382–403.
10. Carver RH, Tai KC. Replay and testing for concurrent programs. *IEEE Software* 1991; **8**(2):86–74.
11. Damodaran-Kamal SK, Francioni JM. Nondeterminacy: Testing and debugging in message passing parallel programs. *3rd ACM/ONR Workshop on Parallel and Distributed Debugging,* New York, 1993; 118–128.
12. Souza SRS, Vergilio SR, Souza PSL, Simao AS, Hausen AC. Structural testing criteria for message-passing parallel programs. *Concurrency and Computation: Practice and Experience* 2008; **20**:1893–1916.
13. Sarmanho FS, Souza PSL, Souza SRS, Simao AS. Structural testing for semaphore-based multithread programs. *International Conference on Computational Science (ICCS 2008), Lecture Notes in Computer Science (LNCS)*, Vol. 5101, 2008; 337–346.
14. Souza SRS, Souza PSL, Machado MCC, Camillo MS, Simao AS, Zaluska E. Using coverage and reachability testing to improve concurrent program testing quality. *23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011),* Eden Roc Renaissance Miami Beach, United States, 2011; 207–212.
15. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation in Software Engineering*. Kluwer Academic Publishers Norwel Springer; 2012 edition (June 17, 2012): MA, USA, 2012.
16. Hwang GH, c Tai K, l Huang T. Reachability testing: an approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering* 1995; **5**:493–510.
17. Souza PSL, Souza SRS, Zaluska E. Structural testing for message-passing concurrent programs: an extended test model. *Concurrency and Computation: Practice and Experience* 2012. doi: 10.1002/cpe.2937.
18. Rapps S, Weyuker EJ. Selecting software test data using data flow information. *IEEE Transaction Software Engineering* 1985; **11**(4):367–375.
19. Hausen AC, Verglio SR, Souza SRS, Souza PSL, Simao AS. A tool for structural testing of MPI programs. *8th IEEE Latin-American Test Workshop (LATW 2007),* Cuzco, Peru, 2007; 1–6.
20. Bonetti Daniel RF, Delbem ACB, Travieso G, Souza PSL. Optimizing van der waals calculi using cell-lists and MPI. *IEEE Congress on Evolutionary Computation*, IEEE: Barcelona, Spain, 2010; 1–7.
21. Quinn MJ. *Parallel computing: Theory and practice*, 2nd. edn. McGraw-Hill: New York, 1994.
22. Grama A, Karypis G, Kumar V, Gupta A. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2003.
23. Krawczyk H, Wiszniewski B. Classification of software defects in parallel programs. *Technical Report 2*, Faculty of Electronics, Technical University of Gdansk: Poland, 1994.
24. DeSouza J, Kuhn B, de Supinski BR, Samofalov V, Zheltov S, Bratanov S. Automated, scalable debugging of mpi programs with intel message checker. *SE-HPCS'05*, ACM: New York, NY, USA, 2005; 78–82.
25. DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. *Computer* April 1978; **11**(4):34–41.
26. Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments. *In Proceedings of the 27th International Conference on Software Engineering (ICSEŠ05), St Louis*, 2005; 15–21.
27. Artho C, Biere A, Honiden S. Enforcer - efficient failure injection. *Lecture Notes in Computer Science (LNCS)* 2006; **4085**:412–427.

28. Chen Q, Wang L, Yang Z, Stoller SD. Have: detecting atomicity violations via integrated dynamic and static analysis. *Lecture Notes in Computer Science (LNCS)* 2009; **5503**:425–439.
29. Chen J. Guided testing of concurrent programs using value schedules. *Ph.D. Thesis*, University of Waterloo, 2009.
30. Christakis M, Sagonas K. Detection of asynchronous message passing errors using static analysis. *Lecture Notes in Computer Science (LNCS)* 2011; **6539**:5–18.
31. Ball T, Burckhardt S, Coons KE, Musuvathi M, Qadeer S. Preemption sealing for efficient concurrency testing. *Lecture Notes in Computer Science*, Vol. 6015: Springer Berlin Heidelberg, 2010; 420–434.
32. Dantas A. Improving developers' confidence in test results of multi-threaded systems: avoiding early and late assertions. *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA),* Orlando, Florida, 2008; 899–900.
33. Dantas A, Gaudencio M, Brasileiro F, Cirne W. Obtaining trustworthy test results in multi-threaded systems. *Technical Report*, Federal University of Campina Grande: Brazil, 2007.
34. Emmi M, Qadeer S, RakamariA Z. Delay-bounded scheduling. *ACM SIGPLAN Notices (POPL 2011)* 2011; **46**(1):411–422.
35. Yu J, Narayanasamy S. A case for an interleaving constrained shared-memory multi-processor. *36th Annual International Symposium on Computer Architecture (ISCA 2009)*, ACM: New York, United States, 2009; 325–336.
36. Kamil A, Yelick K. Enforcing textual alignment of collectives using dynamic checks. *Lecture Notes in Computer Science (LNCS)* 2010; **5898**:368–382.
37. Rungta N, Mercer EG. A meta heuristic for effectively detecting concurrency errors. *Lecture Notes in Computer Science (LNCS)* 2009; **5394**:23–37.
38. Gligoric M, Jagannath V, Marinov D. Mutmut: Efficient exploration for mutation testing of multithreaded code. *Third International Conference on Software Testing, Verification and Validation (ICST),* Paris, France, 2010; 55 –64.
39. Sen A, Abadir MS. Coverage metrics for verification of concurrent systemc designs using mutation testing. *IEEE International High Level Design Validation and Test Workshop (HLDVT),* Anaheim, CA, United States, 2010; 75 –81.
40. Jagannath V, Gligoric M, Lauterburg S, Marinov D, Agha G. Mutation operators for actor systems. *3rd International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2010),* Paris, France, 2010; 157–162.
41. Yang Y, Chen X, Gopalakrishnan G. Inspect: a runtime model checker for multithreaded C programs. *Technical Report*, University of Utah: USA, 2008.
42. Li J, Hei D, Yan L. Correctness analysis based on testing and checking for OpenMP programs. *ChinaGrid Annual Conference (ChinaGrid 09)*: Beijing, China, 2009; 210 –215.
43. Musuvathi M, Qadeer S. Fair stateless model checking. *Programming Language Design and Implementation (PLDI 08)*: Tucson, Arizona, United States, 2008; 362–371.
44. Yang Z, Sakallah K. *SMT-based Symbolic Model Checking for Multi-threaded Programs*. Princeton: NJ, 2008.
45. Aichernig B, Griesmayer A, Schlatte R, Stam A. Modeling and testing multi-threaded asynchronous systems with Creol. *Electronic Notes in Theoretical Computer Science* 2009; **243**:3–14.
46. Aichernig BK, Griesmayer A, Johnsen EB, Schlatte R, Stam A. Conformance testing of distributed concurrent systems with executable designs. *Lecture Notes in Computer Science (LNCS)* 2009; **5751**:61–81.
47. Takahashi J, Kojima H, Furukawa Z. Coverage based testing for concurrent software. *28th International Conference on Distributed Computing Systems Workshops (ICDCS 2008),* Beijing, China, 2008; 533 –538.
48. Sherman E, Dwyer MB, Elbaum S. Saturation-based testing of concurrent programs. *7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009)*, ACM: New York, United States, 2009; 53–62.
49. Yang RD, Chung CG. Path analysis testing of concurrent programs. *Information and Software Technology* 1992; **34**(1):43–56.
50. Yang RD, Chung CG. The analysis of infeasible concurrent paths of concurrent Ada programs. *Fourteenth Annual International Computer Software and Applications Conference (COMPSAC 1990),* Chicago, Illinois, 1990; 424 –429.
51. Yang RD, Chung CG. A path analysis approach to concurrent program testing. *International Phoenix Conference on Computers and Communications,* Scottsdale, Arizona, 1990; 425 –432.
52. Koppol PV, Tai KC. An incremental approach to structural testing of concurrent software. *ACM Sigsoft International Symposium on Software Testing and Analysis (ISSTA 1996)*, ACM: New York, United States, 1996; 14–23.
53. Koppol PV, Carver RH, Tai KC. Incremental integration testing of concurrent programs. *IEEE Transactions on Software Engineering* 2002; **28**(6):607–623.
54. Kojima H, Kakuda Y, Takahashi J, Ohta T. A model for concurrent states and its coverage criteria. *International Symposium on Autonomous Decentralized Systems (ISADS 2009),* Athens, Greece, 2009; 1 –6.
55. Krawczyk H, Wiszniewski B. A method for determining testing scenarios for parallel and distributed software, Technical University of Gdansk: Poland, 1996.
56. Katayama T, Furukawa Z, Ushijima K. Event interactions graph for test-case generations of concurrent programs. *Asia Pacific Software Engineering Conference,* Brisbane, Queensland, Australia, 1995; 29 –37.
57. Katayama T, Furukawa Z, Ushijima K. A method for structural testing of Ada concurrent programs using the event interactions graph. *Asia-Pacific Software Engineering Conference,* Seoul, South Korea, 1996; 355 –364.
58. Katayama T, Furukawa Z, Ushijima K. Design and implementation of test-case generation for concurrent programs. *Asia Pacific Software Engineering Conference,* Taipei, Taiwan, ROC, 1998; 262 –269.

59. Liang Y, Li S, Zhang H, Han C. Timing-sequence testing of parallel programs. *Journal of Computer Science and Technology* 2000; **15**(1):94–95.
60. Lu S, Jiang W, Zhou Y. A study of interleaving coverage criteria. *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, ACM: New York, United States, 2007; 533–536.
61. Kundu S, Ganai MK, Wang C. Contessa: Concurrency testing augmented with symbolic analysis. *Lecture Notes in Computer Science (LNCS)* 2010; **6174**:127–131.
62. Rungta N, Mercer EG, Visser W. Efficient testing of concurrent programs with abstraction-guided symbolic execution. *Lecture Notes in Computer Science (LNCS)* 2009; **5578**:174–191.
63. Tai KC. Reachability testing of asynchronous message-passing programs. In *8th International Conference on Engineering of Complex Computer Systems* (ICECCS 2002), Greenbelt MD (ed.) IEEE Computer Society: USA, 2002; 35–46.
64. Lei Y, Carver RH, Kacker R, Kung D. A combinatorial testing strategy for concurrent programs. *Software Testing Verification and Reliability* 2007; **17**(4):207–225.
65. Carver RH, Lei Y. A general model for reachability testing of concurrent programs. *Lecture Notes in Computer Science (LNCS)* 2004; **3308**:76–98.
66. Carver RH, Lei Y. Distributed reachability testing of concurrent programs. *Concurrency and Computation: Practice and Experience* 2010; **22**(18):2445–2466.
67. Lei Y, Carver R. Reachability testing of semaphore-based programs, 2004; 312–317.
68. Li SQ, Chen HY, Sun YX. A framework of reachability testing for Java multithread programs. *IEEE International Conference on Systems, Man and Cybernetics*, Vol. 3, 2004; 2730 –2734.
69. Carver RH, Lei Y. A stateful approach to testing monitors in multithreaded programs. *IEEE 12th International Symposium on High-Assurance Systems Engineering (HASE)*: San Jose, CA, United States, 2010; 54 –63.
70. Wei W, Wu Y, Lejun Z, Lin G. Testing path generation algorithm with network performance constraints for nondeterministic parallel programs. *Seventh International Conference on Web-Age Information Management Workshops (WAIM 2006)*: Hong Kong, China, 2006; 6.
71. Nagarakatte S, Burckhardt S, Martin MM, Musuvathi M. Multicore acceleration of priority-based schedulers for concurrency bug detection. *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12: Beijing, China, 2012; 543–554.
72. Hong S, Ahn J, Park S, Kim M, Harrold MJ. Testing concurrent programs to achieve high synchronization coverage. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA2012)*, 2012; 210–220.
73. Ding Z, Zhang K, Hu J. A rigorous approach towards test case generation. *Information Sciences* 2008; **178**(21): 4057–4079.
74. Tan RP, Nagpal P, Miller S. Automated black box testing tool for a parallel programming library. *2nd International Conference on Software Testing, Verification, and Validation (ICST 2009),* Denver, Colorado, United States, 2009; 307–316.
75. Xiaoan B, Na Z, Zuohua D. Test case generation of concurrent programs based on event graph. *5th International Joint Conference on INC, IMS, and IDC (NCM 2009),* Seoul, Korea, 2009; 143–149.
76. Souza SRS, Brito MAS, Silva RA, Souza PSL, Zaluska E. Research in concurrent software testing: A systematic review. *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD 2011), in Conjunction with International Symposium on Software Testing and Analysis (ISSTA 2011),* Toronto, ON, Canada, 2011; 1–5.
77. Chung CM, Shih TK, Wang YH, Lin WC, Kou YF. Task decomposition testing and metrics for concurrent programs. *Fifth International Symposium on Software Reliability Engineering (ISSRE 1996),* White Plains, NY, 1996; 122–130.
78. Yang CS, Souter AL, Pollock LL. All-du-path coverage for parallel programs. *International Symposium on Software Testing and Analysis (ISSTA 1998)*, ACM-Software Engineering Notes: Clearwater Beach, Florida, United States, 1998; 153–162.
79. Yang CSD, Pollock LL. All-uses testing of shared memory parallel programs. *Software Testing, Verification and Reliability (STVR)* 2003; **13**(1):3–24.
80. Tasiran S, Keremoğlu ME, Muşlu K. Location pairs: a test coverage metric for shared-memory concurrent programs. *Empirical Software Engineering* June 2012; **17**(3):129–165.
81. Edelstein O, Farchi E, Nir Y, Ratsaby G, Ur S. Multithreaded Java program test generation. *IBM System Journal* 2002; **41**(1):111–125.
82. Yu Jie, Narayanasamy S, Pereira C, Pokam G. Maple: a coverage-driven testing tool for multithreaded programs. *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12: Tucson, Arizona, USA, 2012; 485–502.
83. Gong X, Wang Y, Zhou Y, Li B. On testing multi-threaded Java programs. *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, Vol. 1: Qingdao, China, 2007; 702 –706.
84. Hwang GH, Lin HY, Lin SY, Lin CS. Statement-coverage testing for nondeterministic concurrent programs. *Sixth International Symposium on Theoretical Aspects of Software Engineering (TASE2012),* Beijing, China, 2012; 263 –266.
85. Ratsaby G, Sterin B, Ur S. Improvements in coverability analysis. *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, FME '02, Springer-Verlag: London, UK, UK, 2002; 41–56.