Software Qual J (2018) 26:921–959
DOI 10.1007/s11219-017-9376-4

Contributions for the structural testing of multithreaded programs: coverage criteria, testing tool, and experimental evaluation

Silvana Morita Melo[1] ·
Simone do Rocio Senger de Souza[1] ·
Felipe Santos Sarmanho[1] ·
Paulo Sergio Lopes de Souza[1]

Published online: 19 July 2017

Abstract Concurrent software testing is a challenging activity due to factors that are not present in sequential programs, such as communication, synchronization, and non-determinism, and that directly affect the testing process. When we consider multithreaded programs, new challenges for the testing activity are imposed. In the context of structural testing, an important problem raised is how to deal with the coverage of shared variables in order to establish the association between def-use of shared variables. This paper presents results related to the structural testing of multithreaded programs, including testing criteria for coverage testing, a supporting tool, called ValiPthread testing tool and results of an experimental study. This study was conducted to evaluate the cost, effectiveness, and strength of the testing criteria. Also, the study evaluates the contribution of these testing criteria to test specific aspects of multithreaded programs. The experimental results show evidence that the testing criteria present lower cost and higher effectiveness when revealing some kinds of defects, such as deadlock and critical region block. Also, compared to sequential testing criteria, the proposed criteria show that it is important to establish specific coverage testing for multithreaded programs.

Keywords Multithreaded programs · Shared memory · PThreads · Structural testing · Coverage criteria · Experimental evaluation

✉ Silvana Morita Melo
morita@icmc.usp.br

Simone do Rocio Senger de Souza
srocio@icmc.usp.br

Felipe Santos Sarmanho
fsarmanho@gmail.com

Paulo Sergio Lopes de Souza
pssouza@icmc.usp.br

[1] Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação, ICMC/USP, PO 668, São Carlos (SP), 13560-970, Brazil

1 Introduction

Emerging computing systems are by nature parallel/distributed and ubiquitous, and although discreet, they play an important role in the society. Several modern business applications have used concurrency to improve the use of available resources and the overall system's performance. To be effective, these applications must work correctly and efficiently with multiple interconnected threads that run concurrently.

Non-determinism, synchronization, and communication are some of the features that must be considered during the testing activity. They also pose new challenges for testing and increase the efforts for the correctness evaluation. The non-determinism enables different executions of a program with a single input and production of different and possibly correct outputs. This non-deterministic behavior is due to communication and synchronization of concurrent (or parallel) processes (or threads). The testing activity should identify all the possible synchronization sequences and their related outputs. The deterministic execution techniques can be used to force the execution of a sequence for a given input in the presence of non-determinism in order to analyze the execution result (Lei and Carver 2006).

In the context of multithreaded programs, research has focused on systematic testing techniques, considering mainly issues related to deterministic execution, model checking, coverage criteria, and detection of deadlock and race conditions (Taylor et al. 1992; Yang and Pollock 2003; Wong et al. 2005; Lu et al. 2012; Takahashi et al. 2008; Edelstein et al. 2003; Farchi et al. 2003; Lei and Carver 2006; Carver and Tai 1991; Carver and Lei 2010; Damodaran-Kamal and Francioni 1993; Souza et al. 2012a). These studies have shown relevant improvements for concurrent program testing. However, few works have explored the problem of deriving data flow testing for multithreaded programs (Yang and Pollock 1997). On such programs, the data flow information must correspond to an association between variable definition and use, which usually occurs in different threads. Differently from sequential programs, these associations cannot be known during the static analysis because they depend on the implicit inter-thread communication established in the runtime. Therefore, the data-flow analysis proposed for traditional programs is not completely adequate for this context and adaptations must be explored.

The proposition of new testing techniques and criteria must be appropriately evaluated in relation to their application cost and effectiveness in revealing faults. In the context of traditional program testing, experimental studies have found interesting evidence about the cost, effectiveness, and complementary aspects of data flow testing (Li et al. 2009; Souza et al. 2012b; Xiao et al. 2011; Offutt et al. 1996; Ntafos 1988; Mathur and Wong 1994; Weyuker 1990; Foreman and Zweben 1993; Hutchins et al. 1994). Cost refers to the efforts required for a testing criterion to be satisfied and can be measured by the number of test cases necessary to cover it. Effectiveness refers to the ability of a test set to reveal faults. Complementary aspect (or strength) refers to the ability to cover a testing criterion using a test set adequate to another testing criterion. These factors are important for the proposition of an efficient testing strategy that uses the advantages of each testing criterion. The use of empirical software engineering by the research community is essential in order to collect useful information to guide the selection of testing techniques, and evaluate how usable (effective, efficient, and satisfactory) these techniques are, and estimate their applicability (Höst et al. 2000) in real (industrial) environment.

Contributing to the validation of multithreaded programs, we have proposed a family of structural testing criteria for concurrent programs using the PThread library (POSIX Threads) (Sarmanho et al. 2008). The structural testing criteria were designed to explore information about the control, data, and communication flows of multithreaded programs,

considering both their sequential and parallel aspects. During the static and dynamic analysis, information about the communication and synchronization is obtained and it is used to evaluate the coverage of the testing criteria. Therefore, coverage criteria are useful to evaluate the quality of test cases and, consequently, the quality of the program under testing. Also, these criteria offer a coverage metric that can guide the tester and help him or her to determine if a program has been sufficiently tested.

The evolution of this work generated the tool ValiPthread, which implements the coverage criteria and presents functionalities to support the evaluation of coverage and to deal with the data-flow testing analysis, through a post-mortem methodology (adapted from Lei and Carver 2006).

In order to show the benefits and limitations of our structural testing criteria, this paper presents results of an experimental study conducted to evaluate the cost, effectiveness, and complementary aspect of these testing criteria. Initial results of this study were presented in Melo et al. (2012). Therefore, this paper presents a complete study, following the principles of experimental software engineering (ESE). This study applies the experimental process defined by Wohlin et al. (2000), which includes activities for the definition, planning, conduct, analysis, and packing of the experiment. The fundamentals of ESE are important in order to support the conduct of the experimental study and to guarantee that the results are reliable (Basili 1996). All material generated during the experimental study, including programs, test results, and ValiPthread tool, are available for public access, providing resources to facilitate the conduct of secondary studies, replications, and further comparisons.

The remainder of the paper is organized as follows: Section 2 addresses the related work on concurrent software testing area; Section 3 presents the test model and the structural testing criteria for multithreaded programs. Section 4 presents the ValiPthread testing tool. Section 5 describes the experimental study, including the definition, planning, results, and analysis. Section 6 presents a discussion about the results, including a definition of an application testing strategy and threats to validity related to experimental study. Finally, in Section 7 are summarized the conclusions and future research directions.

2 Related work

This section reviews some proposals related to structural testing techniques for concurrent programs and related to experimental studies in this context. Motivated by the fact that traditional testing techniques are not adequate to test the main aspects of the concurrent/parallel programming, such as non-determinism and interaction among processes/threads, many researchers have developed specific testing criteria for concurrent applications.

Taylor et al. (1992) conducted a pioneer study of the definition of testing methodologies for concurrent programs based on the structural testing principles used in sequential programming (Weyuker 1990).

The proposed testing model introduces a set of structural coverage criteria for concurrent programs based on both a concurrency graph and the notion of concurrent states. The criteria proposed are *all-concurrency-paths, all-proper-concurrency-histories, all-edges-between-concurrency-states, all-concurrency-states* and *all-possible-rendezvous*. Chung et al. (1996) proposed four additional testing criteria for Ada programs, focusing on the rendezvous among tasks: *all-entry-call, all-possible-entry-acceptance, all-entry-call-permutation* and *all-entry-call-dependency-permutation*. The criteria were compared and grouped hierarchically in levels based on their satisfaction difficulty.

Yang and Chung (1992) introduced a path analysis testing of concurrent programs and proposed two models: (1) *task flow graph*, to represent the syntactical view of the task execution behavior and model the task control flow; and (2) *rendezvous graph*, which corresponds to the run-time view. It also models the possible *rendezvous* sequences among distinct tasks. The execution of the program traverses a concurrent path of the rendezvous graph (*C-route*) and a concurrent path of the flowgraph (*C-path*). The authors also developed a controlled execution method to support the debugging activity of concurrent programs. They highlighted three research issues that should be addressed to make their approach practical: C-path selection, test generation, and test execution. A practical application of this model is proposed by Yang et al. (1998). A testing tool called DELLA PASTA (DELaware PArallel Software Testing Aid) is used to support the application of the *all-du-path* criterion. The tool aids in the partial automation of the testing activity by finding all-du-pairs of the parallel code. Although the effectiveness of these testing criteria was not evaluated experimentally, it is one of the first studies on the practical applicability of the structural testing criteria for message passing and shared memory programs. A limitation of this proposal is that only covers shared memory programs with basic synchronization primitives (*wait* and *post*). Our testing criteria consider these primitives and also others communication and synchronization mechanisms, for instance semaphores.

The works of Taylor et al. (1992) and Yang and Chung (1992) inspired us in the proposition of structural testing for message-passing programs (Vergilio et al. 2005; Souza et al. 2008b). 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 (Souza et al. 2012a).

Wong et al. (2005) 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.

Takahashi et al. (2008) propose two testing criteria for concurrent programs: *All-Concurrent-Path* (ACP) and *All-Concurrent-Binominal-Path* (ACBP). They defined a concurrent module flow graph (CMFG) for concurrent programs based on blocks of source code and control flows between blocks. *All-Concurrent-Path* (ACP) criterion is similar to the sequential *All-Paths* criterion and requires the coverage of all possible concurrent paths of the graph. This criterion has exponential cost due to the number of threads of a program. To minimize the problem of explosion in the number of testing cases, the *All-Concurrent-Binominal-Path* (ACBP) criterion is similar to the *All-Edges* criterion; however it defines paths through the block of commands only once.

Lu et al. (2008) and Zhu (1996) proposed a set of coverage criteria based on interleaving among concurrent events. The aim is to minimize the interleaving exploration and consider common models of concurrent faults. The criteria proposed are *all-interleaving*, *all-thread-pair-interleavings*, *all-single-variable-interleaving*, *defined-use*, *pair-interleaving* and *local-or-remote-interleaving*.

Yastrebenetsky and Trakhtenbrot (2011) developed a different approach based on the complexity used by the *McCabe* criterion (McCabe 1976). In the concurrent programming context, the complexity determines the required to cover new edges types; i.e.,

synchronization edges (edges having a synchronization statement). To cover each statement related to synchronization, all corresponding interleaving must be executed at least once. The study evaluates the synchronization statements on the overall complexity of the program, taking into account the sequential and concurrent edges.

Motivated by the ineffectiveness of the stress testing, Musuvathi et al. (2007) developed a tool called *CHESS*, which uses model checking techniques for concurrent programs testing and enables deterministic execution during the test activity for the reproduction of bugs. Differently from other methods that implement deterministic testing, *CHESS* explores non-determinism systematically and consistently using the concept of concurrent test based on scenarios. In this context, a scenario corresponds to unit tests for sequential programs, but considers features from the concurrent program. A scenario is defined by a tester and used by *CHESS* with the model checking technique to systematically generate all possible interleavings, making it possible to reveal and reproduce bugs on multithreaded programs more effectively than exhaustive stress testing.

Edelstein et al. (2002, 2003) 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 (Farchi et al. 2003). 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 non-determinism problem, rather than code coverage and testing criteria.

Souza et al. (2013) present a generic test model that considers both message passing and shared memory paradigms. This model is composed of new testing criteria that explore both of the paradigms. For this, the test model captures information about threads and processes, representing a hierarchy of concurrent tasks, which presents both implicit communication (shared memory) and explicit communication (message passing).

The previous works stress the relevance of providing coverage measures for concurrent and parallel programs, considering basic primitives of interaction of multithreaded programs or the message passing paradigm. Our work is based on the works mentioned above, but differently we explore control and data-flow concepts to introduce criteria specific for the shared memory environment paradigm for programs implemented using the Posix Threads (Pthreads) pattern. These criteria are important in evaluating the quality of test cases as well as in considering that a program has been tested enough.

Despite a variety of testing techniques that have been proposed to concurrent software testing, there are few empirical studies that empirically validate these proposals. We present below some works that propose a new testing technique and evaluate your applicability by means of the application of empirical methods. The effort to apply methods, theories, and techniques from empirical software engineering has as an objective to achieve solid and conclusive findings through rigorous empirical methods and statistical analysis (de Oliveira Neto et al. 2015).

Brito et al. (2013) performed an experiment to evaluate structural testing criteria for message-passing programs, analyze each criterion considering their effectiveness, cost, and the strength. The authors also compared sequential testing criteria and concurrent testing criteria in order to observe which group presents best effectiveness and minor cost.

Hong et al. (2013) conducted an experiment to explore the relationship between concurrent coverage metrics and fault detection effectiveness. In this study, researchers compared concurrent coverage metrics (singular and pairwise). An additional study Hong et al. (2015)

compared the existing concurrency coverage metrics and six new metrics formed by combining complementary metrics (singular, pairwise, and combined). Both of these studies evaluate the impact of concurrency coverage metrics on testing effectiveness and examine the relationship between coverage, fault detection, and test suite size.

Souza et al. (2015b) propose a composite approach that uses reachability testing to guide the selection of the synchronization sequences tests according to a specific structural testing criterion. The researchers present an controlled experiment to evaluate the cost and the effectiveness of the new approach in the context of message-passing concurrent programs developed with Message Passing Interface (MPI).

The empirical studies in concurrent software testing present, in general, less maturity and a lot of challenges needing more attention from the research community in order to develop more mature empirical evidence aiming to gain quantitative insights about the technique's effectiveness in different environments (Melo et al. 2016). In this direction, this work contributes to this scenario, presenting a controlled experiment (Section 6) to validate the structural testing criteria proposed for multithreaded programs. These testing criteria are described in the next section.

3 Structural testing for multithreaded programs

Structural testing is a technique that uses the source code to guide the selection of test data. Data flow testing is a structural testing technique that can be used to detect improper use of data values due to coding faults. In data flow testing, the software under testing is modeled as a control flow graph, allowing to identify the control flow information in the program. In this graph is included information about definition and use of the variables. Based on this information, associations between definitions and uses of the variables are derived and used to guide the application of data flow criteria. Thus, it is possible to design and select the test suite to cover data flow information using a finite number of paths (Badlaney et al. 2006).

Based on these definitions from sequential programming, Sarmanho et al. (2008) proposed a family of structural data and control flow testing criteria for shared-memory concurrent programs. To apply the testing criteria, a test model was established, which is a representation in a graph of the concurrent program designed to collect information about control, data, and communication from these programs.

The model assumes that a fixed and known number of threads, x, is created at the initialization of the concurrent application. Each thread may execute a different source code, however, all of them share a same global memory space. Also, it is assumed that: (1) there is an implicit communication through shared variables; (2) there is explicit synchronization among threads through semaphores, which has two basic atomic primitives: *post* (or *up/unlock*) and *wait* (or *down/lock*); and (3) the initialization and finalization of threads act as a synchronization over virtual semaphores.

To introduce the proposed model and illustrate the application of the testing criteria, the *producer-consumer* program (Fig. 1) is employed. This program was developed in ANSI-C/PThreads and has a concurrent access to a limited buffer through three threads: (1) a *master*, responsible for initializing variables and creating the *producer* and *consumer* threads; (2) a *producer*, to populate a limited buffer; and (3) a *consumer*, to obtain items from the buffer (Sarmanho et al. 2008).

The *master* thread initializes a binary semaphore, called $mutex$, for synchronization, and another two semaphores called $empty$ and $full$ represent slots to be filled or used by the buffer, respectively. After creating the new threads, the *master* thread waits for the execution

Fig. 1 Source code for the producer-consumer program. **a** master thread, **b** producer thread, and **c** consumer thread

(a)

```
sem_t mutex, empty, full;
int queue[2], avail;
int main(void) {
    pthread_t prod_h, cons_h; /*1*/
    avail = 0; /*1*/
    sem_init (&mutex, 0, 1); /*2,3*/
    sem_init(&empty, 0, 2); /*4,5,6*/
    sem_init(&full, 0, 0); /*7*/
    pthread_create(&prod_h, 0,
    (void*) producer, (void*) 0); /*8*/
    pthread_create(&cons_h, 0,
    (void*) consumer, (void*) 0); /*9*/
    pthread_join(prod_h, 0); /*10*/
    pthread_join(cons_h, 0); /*11*/
  exit(0); /*12*/
}
```

(b)

```
void *producer (void){
    int prod, item; /*1*/
    prod = 0; /*2*/
    while (prod < 2) { /*3*/
          item = rand()%1000; /*4*/
         sem_wait(&empty); /*5*/
          sem_wait(&mutex); /*6*/
          queue[avail] = item; /*7*/
          avail++; /*8*/
          prod++; /*9*/
          sem_post(&mutex); /*10*/
          sem_post(&full); /*11*/
  }
    pthread_exit(0); /*12*/
}
```

(c)

```
void * consumer(void) { /*1* /
    int cons = 0, my_item;/*2*/
    while (cons < 2) { /*3*/
        sem_wait(&full); /*4*/
        sem_wait(&mutex); /*5*/
        cons++; /*6*/
        avail--; /*7*/
        my_item = queue[avail]; /*8*/
        sem_post(&mutex); /*9*/
        sem_post(&empty); /*10*/
        printf("%d\n",my_item) }; /*11*/
    }
    pthread_exit(0); /*12*/
}
```

of child threads and its return through primitives $pthread\_join()$ (establishing the end of the threads execution). The *producer* thread is limited to generate at most two items. A new local item is generated and this thread verifies if there are empty slots in the buffer using the primitive $sem\_wait(\&empty)$. If there is at least one $sem\_wait(\&mutex)$ tries to enter in the criticalregion. When *producer* enters it, the item is inserted and the shared variable $avail$ is used and defined(incremented), which signalizes to the *consumer* that there is one more item in the buffer. The implicit communication, from a sender perspective, occurs in this new definition of the $avail$ variable. A similar communication occurs when writing into *the queue* vector. The $prod$ local variable establishes no communication among threads. The *producer* leaves the critical region after its variables have been updated and inserts a token into the $full$ semaphore by means of $sem\_post(\&full)$. This operation signalizes to the *consumer* that another item is in the buffer. On the other hand, the *consumer* thread verifies if at least one item has been inserted in the buffer using primitive $sem\_wait(\&full)$. When the *consumer* tries to reach the critical region, the $avail$ variable is updated and the item is removed from the queue. These two shared variables are responsible for the implicit communication in the *consumer*. The primitive $sem_post(\&mutex)$in the *consumer* thread enables it to leave the critical region and primitive $sem\_post(\&empty)$ signalizes to the *producer* thread there is a new empty slot.

The multithreaded program MT can be specified by a set of k parallel threads: $MT = \{t^0, t^1, ..., t^{k-1}\}$. Each thread t has its own control flow graph CFG^t, composed of a set of nodes N^t and a set of edges E^t (Rapps and Weyuker 1985). Each node n in the thread t is represented using the notation n_i^t and corresponds to a set of commands that are sequentially executed or are associated with a synchronization primitive (*post* or *wait*). In program example (Fig. 1), the numbers between /* and */ represent the nodes in CFG^t of each thread, where t^0 is *master* thread, t^1 is *producer* and t^2 is *consumer* thread.

The complete multithreaded program MT is thus defined by a parallel control flow graph for shared memory $PCFG_{SM}$, which is composed of the individual control flow graphs CFG^t (for $t = 0 \ldots x - 1$) and the representation of the synchronization between the threads. In Fig. 2 is shown the $PCFG_{sm}$ for the *producer-consumer* program, completed with data flow information (the details are described below).

A synchronization edge (n_i^a, n_j^b) links a *post* node in a thread a to a *wait* node in a thread b, in which a and b can be the same thread. By simplification purposes, Fig. 2 shows only some synchronization edges (*s-edges*), represented by dotted lines. For instance, $(3^0, 6^1)$ is a synchronization edge.

We have defined subsets useful for the introduction of testing criteria. N^t and E^t represent the sets of nodes and edges, respectively, in a thread t. N_p^t contains all nodes with *post* primitives and N_w^t carries all nodes with *wait* primitives. E_S represents the set of *s-edges* of MT. N and E represent the sets of nodes and edges of all program MT. For program example, these sets (and others) are illustrated in Table 1.

For a thread t, a path $\pi^t = (n_1^t, n_2^t, ..., n_j^t)$, where $(n_i^t, n_{i+1}^t) \in E^t$, for $1 \leq i < j$, is an intra-thread path if it has no synchronization edges. For instance, in program example, we have the following intra-thread path in t^1: $\pi^1 = (1^1, 2^1, 3^1, 12^1)$.

A path that includes at least one synchronization edge is called an inter-thread path and denoted by $\Pi = (PATHS, SYNC)$, where $PATHS = \{\pi^1, \pi^2, ..., \pi^n\}$. Thus, $(n_i^t, n_j^q) \in SYNC$ indicates the i-th node of the intra-thread path π^t that is synchronized with the j-th node of the intra-thread path π^q. The symbol $\prec$ is used to represent the order of execution between two nodes of different threads. Thus, $n_j^t \prec n_k^q$ represents that the node n_j^t completes its execution before the node n_k^q. In program example,

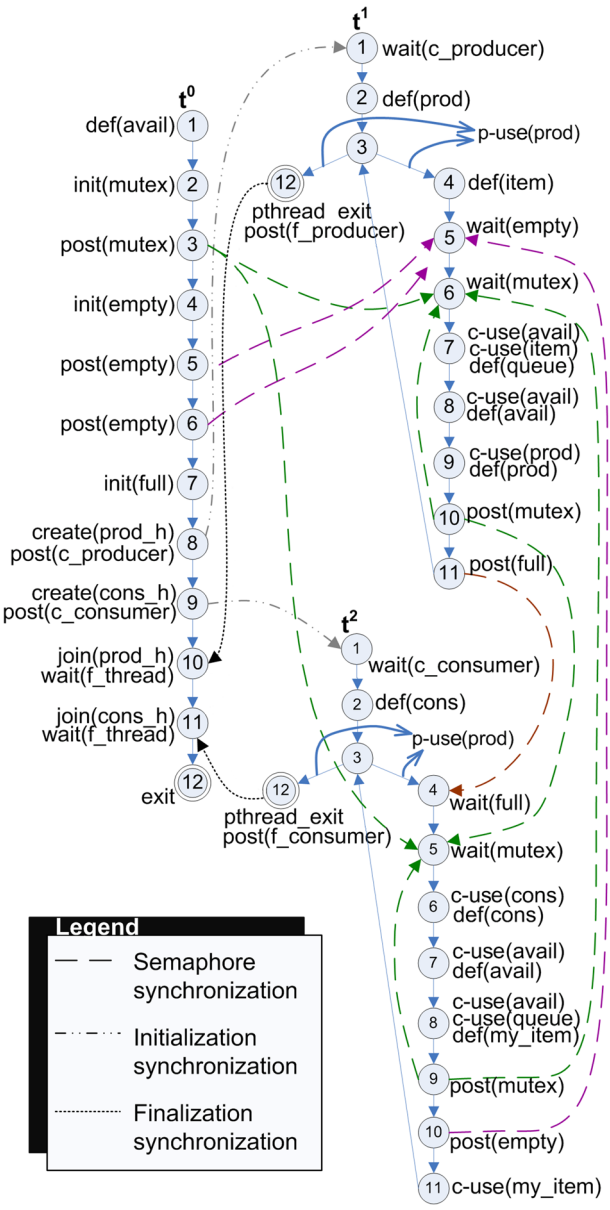


Fig. 2 $PCFG_{SM}$ graph representing the *producer-consumer* program in Fig. 1

we can statically derive, from $PCFG_{SM}$, the following inter-thread path and associated *s-edges*: $\Pi^1 = (1^0, 2^0, ..., 12^0)$, $(1^1, 2^1, 3^1, 4^1, ..., 11^1, 3^1, 12^1)$, $(1^2, 2^2, 3^2, 12^2)$, $((3^0, 6^1), (5^0, 5^1), (6^0, 5^1), (8^0, 1^1), (9^0, 1^2), (10^1, 6^1), (12^2, 11^0))$.

$PCFG_{SM}$ also captures information about data flow. Multithreaded programs have two other special types of variables, apart from local variables, i.e., (1) shared variables, used for communication, and (2) synchronization variables, used by semaphores. Set V denotes

Table 1 Sets of the test model for the *producer-consumer* program in Fig. 1

$n = 3 \qquad MT = \{t^0, t^1, t^2\}$
$N_0 = \{1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 8^0, 9^0, 10^0, 11^0, 12^0\}$
$N_1 = \{1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 7^1, 8^1, 9^1, 10^1, 11^1, 12^1\}$
$N_2 = \{1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2, 10^2, 11^0, 12^0\}$
$N = N_0 \cup N_1 \cup N_2$
$N_p = \{3^0, 5^0, 6^0, 8^0, 9^0, 10^1, 11^1, 12^1, 9^2, 10^2, 12^2\}$
$N_w = \{10^0, 11^0, 1^1, 5^1, 6^1, 1^2, 4^2, 5^2\}$
$E_I^0 = \{(1^0, 2^0), (2^0, 3^0), ..., (10^0, 11^0), (11^0, 12^0)\}$
$E_I^1 = \{(1^1, 2^1), (2^1, 3^1), ..., (11^1, 3^1), (3^1, 12^1)\}$
$E_I^2 = \{(1^2, 2^2), (2^2, 3^2), ..., (11^2, 3^2), (3^2, 12^2)\}$
$E_s = \{(3^0, 6^1), (3^0, 5^2), (10^1, 5^2), (10^1, 6^1), (9^2, 6^1), (9^2, 5^2), (5^0, 5^1), (6^0, 5^1),$
$(10^2, 5^1), (8^0, 1^1), (9^0, 1^2), (11^1, 4^2), (12^1, 10^0), (12^2, 11^0)\}$
$E = E_I^0 \cup E_I^1 \cup E_I^2 \cup E_s$
$M_w(3^0) = \{6^1, 5^2\}$, $M_w(5^0) = \{5^1\}$, $M_w(6^0) = \{5^1\}$,$M_w(8^0) = \{1^1\}$,
$M_w(9^0) = \{1^2\}$, $M_w(10^1) = \{6^1, 5^2\}$, $M_w(11^1) = \{4^2\}$, $M_w(12^1) = \{10^0\}$,
$M_w(9^2) = \{6^1, 5^2\}$, $M_w(10^2) = \{5^1\}$, $M_w(12^2) = \{11^0\}$
$M_p(1^1) = \{8^0\}$, $M_p(5^1) = \{5^0, 6^0, 10^2\}$, $M_p(6^1) = \{3^0, 10^1, 9^2\}$, $M_p(1^2) = \{9^0\}$,
$M_p(4^2) = \{11^1\}$, $M_p(5^2) = \{3^0, 10^1, 9^2\}$, $M_p(10^0) = \{12^1\}$, $M_p(11^0) = \{12^2\}$
$V_L^0 = \{prod\_h, cons\_h\}$, $V_L^1 = \{prod, item\}$, $V_L^2 = \{cons, my\_item\}$
$V_C = \{queue, avail\}$, $V_S = \{mutex, full, empty\}$
$V = V_L^0 \cup V_L^1 \cup V_L^2 \cup V_C \cup V_S$
$def(1^0) = \{avail\}$, $def(8^0) = \{prod_h\}$, $def(9^0) = \{cons_h\}$, $def(2^1) = \{prod\}$,
$def(4^1) = \{item\}$, $def(7^1) = \{queue\}$, $def(8^1) = \{avail\}$, $def(9^1) = \{prod\}$,
$def(2^2) = \{cons\}$, $def(6^2) = \{cons\}$, $def(7^2) = \{avail\}$, $def(8^2) = \{my\_item\}$
$c\text{-}use = (10^0, prod_h), (11^0, cons_h), (7^1, avail, item), (8^1, avail), (9^1, prod),$
$(6^2, cons), (7^2, avail), (8^2, queue, avail), (11^2, my\_item)$
$p\text{-}use = (3^1, 4^1, prod), (3^1, 12^1, prod), (3^2, 4^2, cons), (3^2, 12^2, cons)$

all variables, $V_L^t \subset V$ containing local variables of thread t, $V_C \subset V$ contains the shared variables, and $V_S \subset V$ contains the synchronization variables and $def(n_i^t)$ is a set of variables (local and shared) defined in n_i^t. In Table 1 are illustrated these sets for the program example.

The use of variables on multithreaded programs can be: *computational use (c-use)*, where the use of a variable from V_L^t or V_C^t occurs in a computation statement (a node n^t in $PCFG_{SM}$), associated with an intra-thread path; *predicate use (p-use)*, where the use of a variable from V_L^t or V_C^t occurs in a condition, or predicate, associated with control-flow statement in an intra-thread edge (n^t, m^t) in $PCFG_{SM}$; *synchronization use (sync-use)*, where the use of a variable from V_S occurs in a computation statement; *communicational c-use (comm-c-use)*, where the use of a variable from V_C occurs in a computational statements associated with an inter-thread path; and *communicational p-use (comm-p-use)*, where the use of a variable from V_C occurs in a conditional statements associated with an inter-thread path. In Fig. 2 are denoted the uses and definition of variables. Also, the graph contains information about synchronization and communicational variables of the program: *init, post, wait, join*. In Table 1 is presented the sets related to data-flow information for program *producer-consumer*.

Based on these definitions, two sets of structural testing criteria for multithreaded 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:

- *All-p-nodes (ANP) criterion*: the test set must execute all nodes $n_i^t \in N_p$.
- *All-w-nodes (ANW) criterion*: the test set must execute all nodes $n_i^t \in N_w$.
- *All-nodes (AN) criterion*: the test set must execute all nodes $n_i^t \in N$.
- *All-s-edges (AES) criterion*: the test set must execute all sync edges $(n_i^t, n_j^q) \in E_s$.
- *All-edges (AE) criterion*: the test set must execute all edges $(n_i, n_j) \in E$.

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

- *All-def-comm (ADC) criterion*: the test set must execute paths that cover at least one comm-c-use or comm-p-use association for all definitions of $x \in V_c$.
- *All-def (AD) criterion*: for each n_i^p and $x \in def(n_i^p)$, the test set must execute a path that covers at least one c-use, p-use, comm-c-use or comm-p-use association w.r.t. x.
- *All-comm-c-use (ACCU) criterion*: the test set must execute paths that cover all comm-c-use associations.
- *All-comm-p-use (ACPU) criterion*: the test set must execute paths that cover all comm-p-use associations.
- *All-c-use (ACU) criterion*: the test set must execute paths that cover all c-use associations.
- *All-p-use (APU) criterion*: the test set must execute paths that cover all p-use associations.

For each testing criterion was derived required elements, which represent the minimal data to be covered for the testing criterion to be satisfied. For instance, required elements for the *all-w-nodes criterion* are all nodes with the primitive *wait*. Satisfying a testing criterion may not be possible because infeasible elements may exist in the required element sets. An element is infeasible if there exists no set of values for the parameters (input and global variables) that covers it. The problem of the classification of infeasible elements in software testing is extremely complex and is not discussed in this study. More information about this research topic can be found in Vergilio et al. (2006).

The sets of required elements related to data-flow generated for the *producer-consumer* program are shown in Table 2.

Based on the test model ($PCFG_{SM}$ and criteria definitions), five associations between variable definition and use are established. These associations are useful for definition of data flow testing criteria:

- ***c-use association***: defined by a triple (n_i^t, n_j^t, x) iff $x \in V_L^t$ or $x \in V_C^t$, $x \in def(n_i^t)$, n_j^t, it has a c-use of x and there is at least one definition-clear path in t w.r.t. x from n_i^t to n_j^t.
- ***p-use association***: defined by a triple $(n_i^t, (n_j^t, n_k^t), x)$iff $x \in V_L^t$ or $x \in V_C^t$, $x \in def(n_i^t)$, (n_j^t, n_k^t), it has a p-use of x and there is at least one definition-clear path in t w.r.t. x from n_i^t to (n_j^t, n_k^t).
- ***comm-c-use association***: defined by a triple (n_i^t, n_j^q, x)iff $x \in V_C$, $x \in def(n_i^t)$ and n_j^q, it has a c-use of the shared variable x, for $t \neq q$.
- ***comm-p-use association***: defined by a triple $(n_i^t, (n_j^q, n_k^q), x)$iff $x \in V_C$, $x \in def(n_i^t)$ and (n_j^q, n_k^q), it has a p-use of the shared variable x, for $t \neq q$.

Table 2 Some elements required by the proposed criteria for the *producer-consumer* program

Criteria	Some required elements	Total
All-nodes	1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 8^0, 9^0, 10^0, 11^0, 12^0, 1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 7^1, 8^1, 9^1, 10^1, 11^1, 12^1, 1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2, 10^2, 11^2, 12^2	36
All-p-nodes	3^0, 5^0, 6^0, 8^0, 9^0, 10^1, 11^1, 12^1, 9^2, 10^2, 12^2	11
All-w-nodes	10^0, 11^0, 1^1, 5^1, 6^1, 1^2, 4^2, 5^2	8
All-edges	$(1^0, 2^0)$, $(3^1, 4^1)$, $(3^1, 12^1)$, $(3^2, 4^2)$, $(3^2, 12^2)$, $(3^0, 6^1)$, $(3^0, 5^2)$, $(5^0, 5^1)$, $(6^0, 5^1)$, $(8^0, 1^1)$, $(9^0, 1^2)$, $(10^1, 6^1)$, $(10^1, 5^2)$, $(11^1, 4^2)$, $(12^1, 10^0)$, $(12^1, 11^0)$, $(9^2, 6^1)$, $(9^2, 5^2)$, $(10^2, 5^1)$, $(12^2, 10^0)$, $(12^2, 11^0)$	49
All-s-edges	$(3^0, 6^1)$, $(3^0, 5^2)$, $(5^0, 5^1)$, $(6^0, 5^1)$, $(8^0, 1^1)$, $(9^0, 1^2)$, $(10^1, 6^1)$, $(10^1, 5^2)$, $(11^1, 4^2)$, $(12^1, 10^0)$, $(9^2, 6^1)$, $(9^2, 5^2)$, $(10^2, 5^1)$, $(12^2, 11^0)$	14
All-c-use	$(8^0, 10^0$, prod_h), $(9^0, 11^0$, cons_h), $(2^1, 9^1$, prod), $(4^1, 7^1$, item), $(9^1, 9^1$, prod), $(8^1, 7^1$, avail), $(2^2, 6^2$, cons), $(6^2, 6^2$, cons), $(7^2, 7^2$, avail), $(7^2, 8^2$, avail), $(8^2, 11^2$, my_item)	12
All-p-use	$(2^1, (3^1, 4^1)$, prod), $(2^1, (3^1, 12^1)$, prod), $(9^1, (3^1, 4^1)$, prod), $(9^1, (3^1, 12^1)$, prod), $(2^2, (3^2, 4^2)$, cons), $(2^2, (3^2, 12^2)$, cons), $(6^2, (3^2, 4^2)$, cons), $(6^2, (3^2, 12^2)$, cons)	8
All-comm-c-use	$(1^0, 7^1$, avail), $(7^2, 7^1$, avail), $(1^0, 8^1$, avail), $(7^2, 8^1$, avail), $(1^0, 7^2$, avail), $(8^1, 7^2$, avail), $(1^0, 8^2$, avail), $(8^1, 8^2$, avail), $(7^1, 8^2$, queue)	9
All-comm-p-use	{∅}	0
All-def-comm	$(1^0, 7^1$, avail), $(8^1, 7^2$, avail), $(7^2, 7^1$, avail), $(7^1, 8^2$, queue)	4
All-def	$(1^0, 8^2$, avail), $(2^1, (3^1, 4^1)$, prod), $(4^1, 7^1$, item), $(7^1, 8^2$, queue), $(8^1, 7^1$, avail), $(6^2, (3^2, 12^2)$, cons)	6

The creation of *comm-c-use* and *comm-p-use* associations follows a conservative approach in order to generate required elements that represent all possible communications on a multithreaded program. A complementary approach based on the happens-before relationship (Lei and Carver 2006) is applied after the execution to determine whether the communication edges have been covered.

The casual, or happens-before relationship, denoted by "→", was defined by Lamport (1978) and establishes an execution order among concurrent events. Timestamps are assigned to events of interest by a vector, where, if two events e_1 and e_2 are events in a same thread t^i and e_1 is executed before e_2, the timestamp must indicate e_1 has happened before e_2 ($e_1 \rightarrow e_2$). Similarly, if e_p is a post event in a thread t^i and e_w is an wait event in a thread t^j and if e_p has matched e_w, the timestamp of e_p must indicate e_p has happened before the completion of e_w ($e_p \rightarrow e_w$). Each thread t^i maintains an ordered vector v_i of n timestamps, where n represents a quantity of concurrent threads in execution. The timestamp at position i in v_i is updated when thread t^i has passed for a local event of interest or executes a synchronization event. In the case of synchronization between threads, both

vectors related to threads are updated. Therefore, the happens-before relationship is applied not only to events in a same thread or synchronization events but to types of events. If an event e_1 can affect (or cause) another event e_2, then e_1 happens before e_2 in a casual order, but if neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$, thus e_1 and e_2 are concurrent events (or $e_1||e_2$). Relation $||$ is symmetric, such that $e_1||e_2$ and $e_2||e_1$ are equivalent. The happens-before relation does not allow two concurrent events to be ordered because they can occur in any order. Thereby, it orders partial events in one execution and relation $e_1 \rightarrow e_2$ does not imply $e_2 \rightarrow e_1$.

The communication pairs (definition and use of shared variables) are determined after execution by a postmortem approach that uses information about the program execution collected by means of the parallel program instrumentation, in which each execution traverses a path of $PCFG_{SM}$. This approach is an adaptation of the method proposed by Lei and Carver (2006) and contains the following changes: (1) the LIFO (Last In, First Out) criterion is used when there occurs non-determinism in the possible matching between *post* and *wait*; (2) initialization and finalization events are included to generate timestamps for each thread; and (3) points at which there occur definition and use of shared variables are also considered in the timestamps generation. According to the execution trace, two steps are taken as a result of the executed communication pairs: (1) timestamps are attributed to communication and synchronization events and (2) the timestamps of communication events are compared for the establishment of the happens-before relation.

Each *wait* event has a trace set that represents the *post* events to be synchronized with the *wait* event. Using the LIFO criterion, the latest definition of a variable before its use is considered to establish the pair definition-use of the variables in distinct threads.

Figure 3 shows the trace generated from the execution of the *producer-consumer* program. This trace represents the following execution-sequence: the *master* thread starts executing, defines the $avail$ variable (node 1^0), initializes its semaphores (nodes 2^0 to 7^0) and starts the *producer* thread (node 8^0).

The *producer* thread is scheduled in this moment and synchronizes with *master* thread at *s-edge* (8^0, 1^1). *Producer* accesses the buffer by means of *s-edges* (6^0, 5^1) and (3^0, 6^1). In this moment, semaphore $empty$ has two tokens produced by nodes 5^0 and 6^0; the primitive $sem\_wait(empty)$ in the *producer* thread must choose one of them to consume. The token chooses the last one produced in node 6^0 (according to the criterion LIFO to consume concurrent tokens). When the *producer* thread leaves the critical region, two new posts ($sem\_post()$ primitives) are executed, one for *mutex* 10^1 and another for *full* 11^1.

The operating system may change the context and to schedule the *master* thread, which starts the *consumer* thread producing the token for the virtual semaphore $c\_consumer$ (node 9^0). The *master* thread can be blocked again in $pthread\_join(prod\_h)$ and then waits for the *producer* thread to complete its execution (node 10^0). If the *consumer* thread is schedule in this moment, it consumes the token generated by *master* on node 9^0 (*s-edge* (9^0, 1^2)). It accesses the critical region executing synchronizations (11^1, 4^2), and (10^1, 5^2), consumes one item and leaves the critical region after having generated a new token (9^2 for $mutex$).

$\pi^0 = \{1^0,2^0,3^0,4^0,5^0,6^0,7^0,8^0,9^0,10^0,11^0,12^0\}$
$\pi^1 = \{1^1,2^1,3^1,4^1,5^1,6^1,7^1,8^1,9^1,10^1,11^1,3^1,4^1,5^1,6^1,7^1,8^1,9^1,10^1,11^1,12^1\}$
$\pi^2 = \{1^2,2^2,3^2,4^2,5^2,6^2,7^2,8^2,9^2,10^2,11^2,3^2,4^2,5^2,6^2,7^2,8^2,9^2,10^2,11^2,12^2\}$
SYNCS = $\{(8^0,1^1),(6^0,5^1),(3^0,6^1),(9^0,1^2),(11^1,4^2),(10^1,5^2),(5^0,5^1),(9^2,6^1),(12^1,10^0),(11^1,4^2),(10^1,5^2),(12^2,11^0)\}$

Fig. 3 Traces produced by threads of the program shown in Fig. 1 for the following execution sequence: produced/consumed/produced/consumed

Producer, in this turn, performs its second access to the critical region, inserts a new item into the buffer, leaves the critical region, and finalizes its execution. The synchronizations established are (5^0, 5^1) and (9^2, 6^1); new tokens are produced for semaphores $mutex$, $full$ and *f_producer* (the latter is the virtual semaphore to release *pthread_join*(*prod_h*) (node 10^0). The *master* thread then can be scheduled, executes the synchronization (12^1, 10^0), and is blocked again on node 11^0. The *consumer* thread when it is scheduled again, generates a token for $empty$ on 10^2, accesses the critical region, executes synchronizations (11^1, 4^2) and (10^1, 5^2), consumes the second item from the buffer, leaves the critical region after having produced two new tokens (9^2 for $mutex$ and 10^2 for $empty$) and finalizes its execution producing token 12^2 for the virtual semaphore *f_consumer*. The *master* thread comes back to the execution and performs the synchronization (12^2, 11^0) releasing *pthread_join*() in the *master* for the *consumer*. In this moment, all three of the threads have been completed.

Figure 4 shows a diagram of logical space-time with timestamps for the uncontrolled execution showed in Fig. 3 and the variables and definitions illustrated in Fig. 1. These diagrams are used for obtaining the definition and using pairs of shared variables among the distinct threads exercised during the execution (i.e., the *comm-c-use* and analogously *comm-p-use* associations covered). Therefore, for the variable $avail$, there are five *comm-c-use*

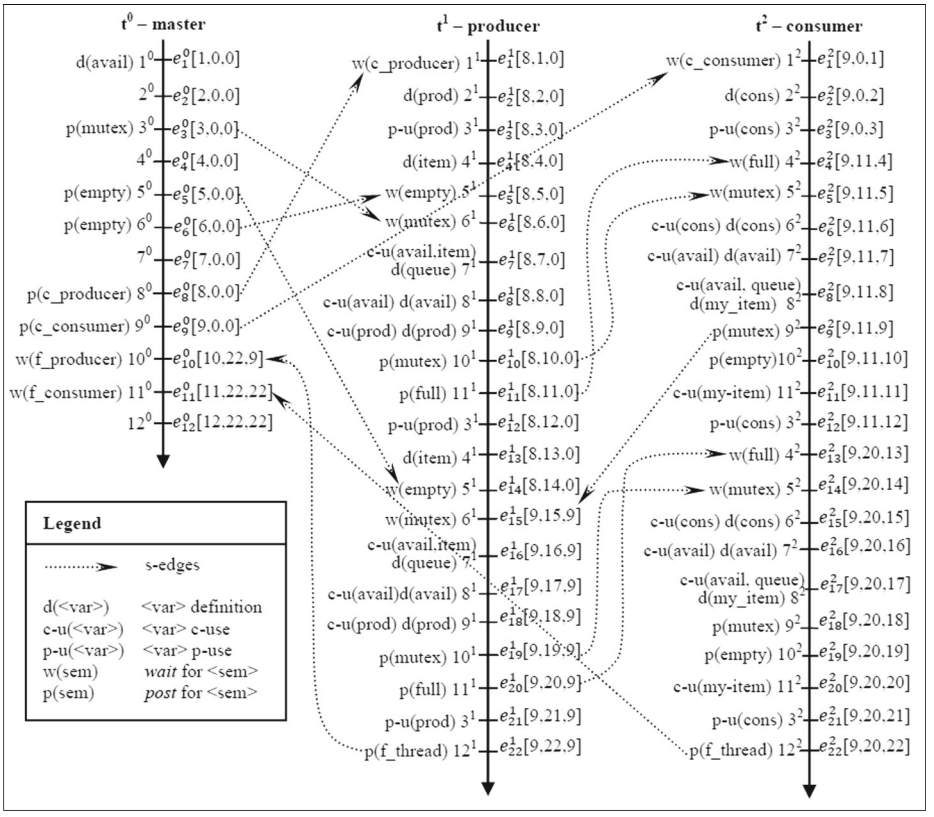


Fig. 4 Logical space-time diagram based on timestamps method for the *producer-consumer* program, showing the first execution for the sequence: produce-consume-produce-consume. On the *left side* of each vertical time-line is the operation performed and its associated node. The *right side* shows event i associated which thread t (using notation e_i^t) and the respective vector of timestamps

associations covered (($1^0, 7^1$), ($1^0, 8^1$), ($8^1, 7^2$), ($7^2, 7^1$) and ($7^2, 8^1$)) and three associations not covered ($1^0, 7^2$), ($1^0, 8^2$), and ($8^1, 8^2$) of a total of eight possibilities. The *comm-c-use* association ($7^1, 8^2$) was covered by the shared variable *queue*.

The application of the timestamp approach enable the obtaining of race sets $e^1_{14} = \{e^2_{10}\}$, $e^1_{15} = \{e^1_{10}\}$ and $e^2_{14} = \{e^2_9\}$ for primitives *sem_wait*(). The race sets for other events with primitives *sem_wait*() are empty.

Two new controlled executions were conducted with the primitives *sem_post*() existent in the race sets, t. The first execution considers the race set of the event e^1_{14}, synchronizing now with e^2_{10} (a *sem_post*(*empty*) on *consumer*). This change produced new pairs of synchronizations, which were executed in a non-controlled way. The trace file for this new execution is shown in Fig. 5. This procedure is repeated until all the possible synchronizations have been executed so as to improve the coverage of *comm-c-use* associations.

4 ValiPthread testing tool

The ValiPThread software testing tool was implemented to support the effective application of the testing criteria defined in the previous section (Sarmanho et al. 2008). It provides functionalities to (i) create test sessions, (ii) save and execute test data, and (iii) evaluate the testing coverage w.r.t a given testing criterion. The architecture of ValiPThread is based on the ValiPar testing tool (Souza et al. 2005) for parallel programs developed for message passing and contains the adaptations necessary for the context of multithreaded programs. It consists of four main modules: *ValiPthread_Inst*, responsible for the static analysis of parallel programs and generation of a program instrumented version; *ValiPthread_Elem*, which generates the list of required elements for the defined criteria; *ValiPthread_Exec*, responsible for the parallel program execution, generation of the executed paths, and assistance in the deterministic execution; and *ValiPthread_Eval*, responsible for attributing timestamps to events on the program and evaluating the coverage obtained. The architecture of ValiPThread is shown in Fig. 6.

ValiPThread performs the multithreaded program test based on the $PCFG_{sm}$ model and following four main steps:

1. **obtaining of elements from a static analysis**: information about the execution of the program is obtained from the source code and inserted in $PCFG_{sm}$ to be used later to guide the testing activity. An instrumented version of the concurrent program is produced from the static analysis (by IDEL tool Simao et al. 2003) and used to generation of trace files;
2. **definition of the required elements**: required elements are generated based on both test model $PCFG_{sm}$ and the criteria;

$\pi^0 = \{1^0,2^0,3^0,4^0,5^0,6^0,7^0,8^0,9^0,10^0,11^0,12^0\}$
$\pi^1 = \{1^1,2^1,3^1,4^1,5^1,6^1,7^1,8^1,9^1,10^1,11^1,3^1,4^1,5^1,6^1,7^1,8^1,9^1,10^1,11^1,12^1\}$
$\pi^2 = \{1^2,2^2,3^2,4^2,5^2,6^2,7^2,8^2,9^2,10^2,11^2,3^2,4^2,5^2,6^2,7^2,8^2,9^2,10^2,11^2,12^2\}$
SYNCS = $\{(8^0,1^1),(6^0,5^1),(3^0,6^1),(9^0,1^2),(11^1,4^2),(10^1,5^2),(10^2,5^1),(9^2,6^1),(12^1,10^0),(11^1,4^2),(10^1,5^2),(12^2,11^0)\}$

Fig. 5 Traces produced for the second execution of the example program showing the items (produced), (consumed), and (produced and consumed). In this trace, unlike of the execution shown in Fig. 3, the *s-edge*($10^1, 5^2$) was coverage rather than *s-edge*($10^1, 5^0$)

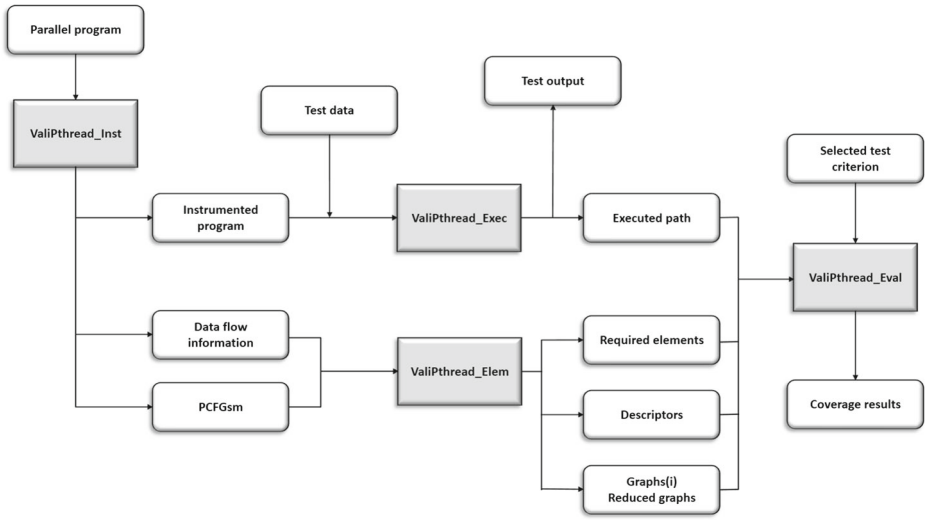


Fig. 6 ValiPThread tool architecture

3. **definition and execution of test cases**: a coverage criterion is established and the instrumented program is run against each test case selected to cover specific required elements;
4. **analysis of the coverage achieved**: the coverage analysis is based on the coverage rate achieved by the application of test cases and guided by a criterion.

The next subsections describe the execution of the multithread programs test divided among the four modules of ValiPthread tool.

4.1 ValiPThread_Inst

The ValiPThread_Inst module is responsible for the source code analysis and instrumentation. It receives the source code as input and generates the $PCFG_{sm}$, data flow information and the instrumented program as output. The parsing performed gathers the elements from the source code to fill them in the sets used by $PCFG_{sm}$, which is based on information about nodes, edges, definition and use of local and shared variables. The instrumented program is used by ValiPThread_Exec to generate traces of the execution responsible for determining the paths executed in the graph and obtaining synchronization sequences during the execution.

4.2 ValiPThread_Elem

The ValiPThread_Elem module generates the elements required for covering the criteria based on $PCFG_{sm}$ and data flow information (both previously generated by ValiPThread_Inst). It also generates auxiliary descriptors and graphs (graph (i) and the inheritors reduced graph) used by ValiPThread_Eval to check whether the elements were covered when the test cases were run.

4.3 ValiPThread_Exec

The ValiPThread_Exec module receives the instrumented program from ValiPThread_Inst and the test data given by the user as input when the execution of the program was requested.

During the execution of the instrumented program, ValiPThread_Exec registers the test case, the execution parameters and an execution trace file for each thread.

The *trace* files contain the executed nodes of each thread that correspond to the test case path. Each node is generally represented by the notation *node-thread*. When a node represents a *post* or *wait* primitive, a third number, representing the event number for that node is defined. Therefore, when this node is inside a repetition structure, it is possible to determine which execution of that node has matched a respective *wait* or *post*. When a node represents a *wait* primitive, it follows notation *wait node post node wait node*, which facilitates the task of the ValiPThread_Eval module when recovering the information from the trace files. The first *wait* represents the cases in which a *wait* node is executed, but a *post* node cannot be reached. Observe that the trace file, for a *post* node, does not represent the *wait* node reached during the execution; such information is registered by the *wait* node.

The tester can examine the execution outputs and the trace file to verify whether the output obtained is correct. If it is wrong, a fault is identified and a debug activity can be performed to reveal the position of the defect in the source code.

At the end of the execution, the Valipthread_Exec module create a subdirectory with information about the executed test case, and *traces* files for each thread, order of synchronization events for each semaphore, file *args* with the register of inputs provided, file *stdout* with the register output achieved by the program, file *stdin* of the registration provided by the user as default input, and *nthreads* with the number of threads created and/or executed by the program.

ValiPThread_Exec also enables the controlled execution of the concurrent program under testing. The controlled execution guarantees two executions of the concurrent program with the same input will produce the same paths and synchronization sequences. This re-execution is useful for replaying the test activity. The controlled execution implemented is based on Lei and Carver (2006). Synchronization sequences are generally gathered from both trace files and synchronization event files for each semaphore.

4.4 ValiPThread_Eval

The ValiPThread_Eval module is responsible for evaluating the coverage of test criteria. It receives as input the trace files produced in the ValiPThread_Exec module, the required elements, and the reduced graph of heirs (both from ValiPThrad_Elem) and evaluates the coverage of the criteria. The percentage of criteria coverage and a list of non-covered elements are provided as output of the module.

4.5 Testing procedures with ValiPThread

ValiPThread and the proposed criteria can be applied following two basic procedures: (1) guidance of the selection of test cases to the program, and (2) evaluation of the test set quality in terms of code and communication coverage.

1. **Test Data Selection Procedure**:
   The following steps should be taken during the procedure:
   (a) Choice of a testing criterion to guide the test data selection (the tester chooses a testing criterion to generate required elements);
   (b) Identification of the test data that cover the elements required for the testing criterion;

(c) Analyses of the correctness of the output for each test case. If the output is not correct, the program must not be corrected;
(d) Identification of new test cases that cover new required elements if there still exists elements to be coverage;
(e) Repetition of the method until the desired coverage has been obtained (ideally 100%). Other testing criteria may be selected to improve the quality of the generated test cases.

In some cases, the existence of infeasible elements does not enable a 100% coverage of a criterion. The determination of infeasible elements is an undecidable problem (Frankl and Weyuker 1986), therefore, the tester must determine the infeasibility of the paths and required elements manually.

2. **Test Data Evaluation Procedure**:
The following steps should be taken for the evaluation of a test set T or comparison between two test sets T_1 and T_2:

(a) Execution of the program with all test cases of T (or T_1 and T_2) for the generation of the trace files and further execution data;
(b) Selection of a testing criterion and evaluation of the coverage of T (or the coverage of T_1 and T_2);
(c) Generation of new test data by the tester if the coverage obtained is not the one expected;
(d) Creation of a test session for each test set and comparison with the coverage obtained.

Notice that these procedures are not exclusive. If an ad hoc test set is available, it can be evaluated according to procedure 2. If the obtained coverage is not adequate, this set can be improved by procedure 1. The use of such an initial test set reduces the efforts for the application of the criteria, therefore, our criteria can be considered complementary to *ad hoc* approaches. They can improve the efficacy of the test cases generated by ad hoc strategies and offer a coverage measure for their evaluation. This metric can be used to measure if a program was tested enough and the testing can be finished.

5 Experimental study

This section reports the definition, design, execution, and analysis of the results of the experimental study conducted in order to evaluate the testing criteria for multithreaded programs. The experimental study was developed following the process for experimental software engineering defined by Wohlin et al. (2000). In this study we evaluated the cost, effectiveness, and strength of the coverage criteria, providing information for further secondary studies. During the experimental study, we have evaluated the testing criteria comparing sequential testing criteria against concurrent testing criteria. The objective was to collect evidence about the benefits of the coverage testing criteria specifics for concurrent programs.

5.1 Experimental planning

In this phase, the foundation for the experiment conduct is defined. The context of the experiment is established in a detailed way. Moreover, the hypothesis of the experiment is stated formally, including null hypotheses and corresponding alternative hypotheses.

5.1.1 Context selection

All execution of the experiments use the ValiPthread testing tool with the following system configuration setting: GNU/Linux operational system using the Ubuntu 11.10 distribution with 3.0.0-32-generic-pae, 4.1.2 gcc compiler.

5.1.2 Goal definition

The goal of this study is analyze the structural testing criteria for multithreaded programs (presented in Section 3) for the purpose of evaluation with respect to their application cost, effectiveness, and strength from the point of view of the researcher in the context of testing specialists finding defects in concurrent code.

Based on this objective, the following research question has been defined:

What are the effectiveness, application cost, and strength of the testing criteria for multithreaded programs?

5.1.3 Hypotheses formulation

Based on such a question, the following hypotheses have been formulated:

Null Hypothesis 1 (NH1): The cost is the same for all concurrent testing criteria (CC) and sequential testing criteria (SC). The cost is measured by means of the metrics described in Section 5.2.

NH1: cost(CC) = cost(SC).

Alternative Hypothesis 1.1 (AH1.1): The cost is different for at least one concurrent testing criteria (CC) compared with sequential testing criteria (SC).

AH1.1: cost(CC) $\neq$ cost(SC).

Null Hypothesis 2 (NH2): The effectiveness is the same for all concurrent testing criteria and sequential testing criteria. The effectiveness is measured by means of the metrics described in Section 5.2.

NH1: effectiveness(CC) = effectiveness(SC).

Alternative Hypothesis 2.1 (AH2.1): The effectiveness is different for at least one for at least one concurrent testing criteria (CC) compared with sequential testing criteria (SC).

AH2.1: effectiveness(CC) $\neq$ effectiveness(SC).

Null Hypothesis 3 (NH3): Regarding strength, no testing criterion subsumes another. The strength is evaluated by means of the metrics described in Section 5.2.

NH3: TCS_{ci}(test case set for testing criteria i) $\not\subset$ TCS_{cj}(test case set for testing criteria j).

Alternative Hypothesis 3 (AH3): There is at least one multithreaded testing criterion that subsumes another.

AH3: TCS_{ci}(test case set for testing criteria i) $\subset$ TCS_{cj}(test case set for testing criteria j).

5.1.4 Objects definition

A set of 33 concurrent programs found on the literature was selected. Such programs have been widely used for the evaluation of new testing techniques in the context of multithreaded programs (Yang et al. 2008; Wang et al. 2009; Jalbert and Sen 2010; Cordeiro and Fischer 2011; Wesonga et al. 2011; Mühlenfeld and Wotawa 2007). We consider them representative for our context, as they enable a comparison with other techniques proposed. Another benefit is that they avoid the introduction of bias that might occur when the researcher is

developing his or her programs according to aspects evaluated in the studies. The programs selected for this study are:

1. Five classical programs that implement solutions for commonly concurrent problems (Tanenbaum 1995; Grama et al. 2003);
2. Programs from Inspect benchmark (12 programs) (Yang 2014);
3. Programs from Helgrind benchmark (12 programs) (Valgrind-Developers 2014);
4. Programs from Rungta benchmark (4 programs) (Rungta and Mercer 2009).

Table 3 shows the complexity of the programs regarding number of code lines (LOC), number of threads, number of *post* and *wait* primitives and number of *synchronization edges (S-edges)*. These features expose specific features related to concurrency and they can interfere in the cost of the test activity. In concurrent software, the complexity of a program is mainly related to the way in which the processes and threads communicate and synchronize.

Despite the ability of data flow testing in detecting data interaction faults, a big gap between real-world programs and the practicality of proposed data flow techniques still exists (Weyuker 1990; Denaro et al. 2013). Studies in the area of sequential software testing (Mathur and Wong 1993; Weyuker 1990; Foreman and Zweben 1993; Offutt et al. 1996) has investigated the effectiveness of data-flow testing employing a limited set of sequential programs that cannot be scalable to large applications. In the context of concurrent applications, the challenge of apply data flow criteria to wide application in industry practice remains and increase with the path-explosion problem result from the non-determinism. We try to avoid this threat by using well-known benchmarks, improving the replication, reuse, and re-analysis.

5.1.5 Variable selection

The variable selection defines the influences and effects, which will be examined in the experiment. Two types of variables were defined in this experiment:

A. *Independent variables*
   Independent variables are the factors that frequently variate in the experiment to measure their effect on the dependent variables. Independent variables for this study include the structural testing criteria for multithreaded programs, the approach used to create test case sets, and the approach used to inject faults in the programs.
B. *Dependent variables*
   The dependent variable is the characteristic evaluated to measure the influence of the factors on the treatments in the experiment. As dependent variables to this study we evaluate the cost, effectiveness, and strength of the multithreaded testing criteria.

5.2 Experiment operation

The experimental setup was defined after the selection of the experiment variables. The setup determines how the experiment will be conducted and, in this case, the experiment was conducted in five steps as described below

A. *Generation of test case sets adequate to the testing criteria*
   A test case set T_i was manually generated for each program based on testing criteria. Using the ValiPThread tool, T_i was executed and the coverage for all criteria was evaluated. Additional test cases were then generated to cover all feasible required elements of each testing criterion. Some programs do not require input data to be executed,

Table 3 Set of concurrent programs used in the experimental study

Programs	LOC	Threads	Posts	Waits	S-edges	Inserted faults	Type of faults
Sleeping barber	62	4	4	4	67	7	1 (1), 2 (1), 4 (3), 8 (2)
GCD	101	4	11	11	81	5	2 (3), 4 (2)
Jacobi	192	5	12	9	189	7	1 (2), 2 (2), 4 (3)
MMult	148	13	4	4	444	6	1 (2), 2 (2), 4 (2)
ProdCons	97	5	5	4	16	7	2 (3), 4 (3), 8 (1)
Carter01	45	3	6	6	32	8	1 (2), 2 (2), 4 (2), 6 (2)
Deadlock01	30	3	4	4	18	4	1 (3), 2 (1)
Dpor-example1	73	4	3	3	12	3	1 (1), 2 (2)
Dpor-example2	104	4	9	6	24	4	1 (1), 2(3)
Dpor-example3	30	4	0	0	24	4	1 (1), 2 (1), 4 (2)
Lazy01	39	4	3	3	24	4	1 (1), 2 (1), 4 (1), 8 (1)
Phase01	25	3	4	4	48	4	1 (2), 2 (2)
Race01	15	3	0	0	14	2	2 (1), 7 (1)
Rafkind01	15	2	1	1	4	3	2 (1), 4 (1), 8 (1)
Simple1	27	3	2	2	12	5	2 (2), 4 (1), 7 (2)
Stateful01	34	3	4	4	26	2	4 (1), 7 (1)
Stateful06	40	3	2	2	12	4	2 (1), 8 (3)
Hg01_all_ok	21	3	2	2	20	2	1 (1), 8 (1)
Hg02_deadlock	30	3	5	5	24	4	1 (2), 4 (1), 8 (1)
Hg03_inherit	41	3	0	0	12	3	2 (3)
Hg04_race	16	3	0	0	14	3	2 (2), 4 (1)
Hg06_readshared	21	3	0	0	6	3	4 (1), 8 (2)
Tc01_simple_race	19	2	0	0	7	4	2 (2), 4 (1), 6 (1)
Tc02_simple_tls	20	2	0	0	3	3	2 (1), 4 (1), 8 (1)
Tc05_simple_race	28	2	2	2	13	3	2 (2), 4 (1)
Tc06_two_races	30	2	2	0	19	4	2 (3), 8 (1)
Tc13_laog	32	1	4	4	0	3	2 (2), 4 (1)
Tc16_byterace	25	1	0	0	1	3	2 (2), 7 (1)
Tc18_semabuse	26	1	1	1	3	4	2 (3), 4 (1)
Account	91	3	6	6	18	8	2 (6), 4 (2)
Reorder	66	5	0	0	60	4	2 (4)
Twostage	82	3	5	4	22	7	2 (3), 4 (4)
Wronglock	67	3	5	4	12	10	2 (8), 4 (1), 7 (1)

but they exhibit a non-deterministic behavior. Therefore, they should be executed several times until the coverage of all required elements (excluding infeasible ones) so that different synchronization sequences can be obtained. In such cases, the controlled execution was used to improve the coverage of the synchronization sequences.

In this step, the infeasible elements were detected and manually cataloged by the tester. Table 4 shows the cost for each testing criteria, related to the size of the adequate test set T_i for each benchmark program. Table 5 shows the number of covered elements and infeasible elements of each program. The symbol $(*)$ represents that it was not possible to generate required elements for a specific criterion.

Table 4 Number of test cases necessary to satisfy each testing criterion

Program	AN	ANP	ANW	AE	AES	ACU	APU	ASU	ACCU	ACPU
Sleeping barber	1	1	1	16	16	1	1	16	1	1
GCD	8	3	5	17	14	7	8	12	7	2
Jacobi	4	2	3	11	11	2	2	14	6	5
Mmult	12	12	12	52	52	12	12	52	24	*
ProdCons	2	1	1	6	6	2	1	7	7	2
Carter01	1	1	1	5	5	1	*	5	1	1
Deadlock01	1	1	1	2	2	1	*	2	2	*
Dpor-example1	1	1	1	2	2	2	2	2	1	2
Dpor-example2	1	1	1	4	4	1	1	4	1	1
Dpor-example3	1	1	1	3	3	1	*	3	3	*
Lazy01	1	1	1	4	4	1	*	4	2	3
Phase01	1	1	1	7	7	1	*	7	*	*
Race01	1	1	1	2	2	1	*	2	2	*
Rafkind01	1	1	1	1	1	1	*	1	*	*
Simple1	1	1	1	2	2	1	*	2	2	*
Stateful01	1	1	1	5	5	1	*	5	2	*
Stateful06	1	1	1	2	2	1	1	2	1	*
Hg01_all_ok	1	1	1	2	2	1	*	2	2	*
Hg02_deadlock	1	1	1	2	2	1	*	2	*	*
Hg03_inherit	1	1	1	2	2	1	*	2	*	*
Hg04_race	1	1	1	2	2	1	*	2	2	*
Hg06_readshared	1	1	1	1	1	1	*	*	*	*
Tc01_simple_race	1	1	1	2	2	*	1	2	2	*
Tc02_simple_tls	1	1	1	1	1	*	1	1	1	*
Tc05_simple_race	1	1	1	2	2	*	1	2	2	2
Tc06_two_races	1	1	1	2	2	*	1	2	2	2
Tc13_laog1	1	*	*	1	*	1	*	*	*	*
Tc16_byterace	1	1	1	1	1	1	1	*	*	*
Tc18_semabuse	1	1	1	1	1	*	*	1	*	*
Account	1	1	1	5	5	1	1	5	1	3
Reorder	1	1	1	21	19	1	1	19	*	6
Twostage	4	3	1	4	2	2	3	4	1	2
Wronglock	3	2	1	4	2	2	3	3	2	1
Average	2	2	2	6	6	2	2	6	3	2

B. *Generation of the programs with injected faults*

In order to evaluate the effectiveness, faults were inserted on the programs. The inserted faults were inspired in the faults taxonomy for multithreaded programs presented by Bradbury and Jalbert (2009) (Table 6).

The faults are injected manually in the program code by graduate students in computer science. The subjects have previous knowledge in software testing and concurrent programming. The faults are inserted by the subjects following the taxonomy adopted

Table 5 Number of covered/infeasible elements for each testing criterion

Program	AN	ANP	ANW	AE	AES	ACU	APU	ASU	ACCU	ACPU
Sleeping barber	82/6	29/5	19/0	76/14	67/14	27/0	8/2	57/13	2/0	4/1
GCD	161/6	27/4	25/0	130/22	81/18	44/7	118/15	60/14	39/10	8/4
Jacobi	283/27	40/9	27/2	273/130	189/118	68/1	78/16	243/158	349/256	308/247
Mmult	353/13	66/41	49/0	539/236	444/236	247/0	236/27	404/228	448/128	*
ProdCons	41/0	11/0	8/0	21/5	16/5	8/0	8/2	10/1	13/3	*
Carter01	44/3	13/0	10/0	42/19	32/14	2/0	*	26/10	18/14	36/30
Deadlock01	24/0	10/0	8/0	21/6	18/6	2/0	*	12/4	4/2	*
Dpor-example1	69/4	8/2	5/0	26/6	12/4	36/2	32/9	12/4	2/0	8/4
Dpor-example2	66/9	15/4	7/0	37/16	24/10	36/8	28/12	22/10	2/2	20/0
Dpor-example3	30/0	12/0	10/0	28/10	24/10	3/0	*	12/4	6/3	*
Lazy01	27/1	11/0	9/0	28/9	24/9	3/0	*	12/3	3/1	3/0
Phase01	30/0	14/0	12/0	51/22	48/22	2/0	*	42/17	*	*
Race01	18/0	7/0	6/0	17/4	14/4	2/0	*	8/3	4/2	*
Rafkind01	11/0	4/4	3/3	6/1	4/1	1/0	*	2/1	*	*
Simple1	19/0	7/0	6/0	15/4	12/4	2/0	*	6/2	4/2	*
Stateful01	28/0	10/0	8/0	29/12	26/12	2/0	*	20/8	6/2	*
Stateful06	26/0	7/0	6/0	17/4	12/4	8/0	8/2	6/0	3/2	*
Hg01_all_ok	20/0	8/0	7/0	23/9	20/9	2/0	*	14/6	4/2	*
Hg02_deadlock	24/0	11/0	9/0	27/11	24/11	2/0	*	18/9	*	*
Hg03_inherit	33/4	7/0	6/0	20/8	12/4	7/0	*	6/2	*	*
Hg04_race	18/0	7/0	6/0	17/4	14/4	2/0	*	8/3	4/2	*
Hg06_readshared	14/0	4/0	4/0	9/2	6/2	3/0	*	*	*	*
Tc01_simple_race	16/12	6/1	3/0	10/3	7/2	*	2/1	6/2	4/2	*
Tc02_simple_tls	15/2	5/1	2/0	6/2	3/1	*	2/1	2/1	9/7	*
Tc05_simple_race	27/4	9/1	5/0	18/6	13/4	*	2/1	12/4	12/8	8/4
Tc06_two_races	35/4	12/1	7/0	24/8	19/6	*	2/1	18/6	18/12	12/6
Tc13_laog1	26/2	2/2	*	1/0	*	10/0	*	*	*	*
Tc16_byterace	24/0	3/0	1/0	9/1	1/0	16/0	18/5	*	*	*
Tc18_semabuse	13/1	5/1	2/0	5/2	3/2	*	*	2/1	*	*
Account	57/5	11/0	7/0	27/5	18/3	26/3	18/6	12/3	15/12	18/12
Reorder	69/5	17/0	14/0	77/15	60/13	26/0	28/4	48/6	*	48/24
Twostage	57/6	14/3	6/0	35/14	22/13	20/0	40/7	22/15	6/3	4/2
Wronglock	46/1	9/0	6/0	23/5	12/4	20/0	38/5	8/2	9/3	6/5
Average	55/4	13/3	9/0	51/19	40/18	22/1	39/7	38/18	41/21	37/26

(Bradbury and Jalbert 2009). The resulting relation between the type of faults and number of defects inserted in each program is shown in Table 3. The seventh column of the table (inserted faults) refers to the total of versions from original program with one seeded fault, which is evaluated separately during the testing activity. The eighth column of the table (type of faults) refers to the number of faults inserted in each program by the specific type of fault inserted. The first number represents the type of fault (as defined in Table 6) and the number in parentheses represents the number of faults inserted for this type. All data used and generated during the empirical study, including

Table 6 Faults taxonomy (Bradbury and Jalbert 2009) used to inject faults in the experimental programs

Fault type	Definition
1. Deadlock	Occurs when a process is unable to proceed because it is waiting for another process that is locked, forming a cycle (Tanenbaum 1995).
2. Wrong lock or no lock	Occurs when a thread does not obtain the same lock instance during the execution or when a new incorrect lock has been added.
3. Interference	Occurs when two or more concurrent threads access a shared variable and the last access is write. In this case, the threads non-use the explicit mechanism to prevent concurrent access.
4. Incorrect count initialization	Occurs when there exists an incorrect initialization in a barrier or in the number of semaphore permissions.
5. Resource exhaustion	Occurs when a group of threads holds all of finite number of resources and another thread needs additional resources, but none gives a up.
6. Blocking critical section	Occurs when a thread takes control of the critical region to eventually return but it never does.
7. Non atomic operations assumed to be atomic	Occur when an operation is considered atomic by the programmer. In lower abstraction levels, they consist of several non-atomic unprotected operations.
8. The sleep	Occurs when the programmer inappropriately adds a sleep method to the parent thread to coordinate child thread, but the correct would use the join method.
9. Missing or nonexistent signals	Occurs when a,thread remains locked, because the notification (unlock signal) is missing or does not exist.
10. Missing a notify	Occurs when a *notify()* is executed before its corresponding *wait()*, and is missed.

programs, fault programs, testing tool, and results are available online,[1] facilitating systematic reuse (e.g., replication, systematic review, and meta-analysis).

C. *Evaluation of the application cost*

The cost was analyzed based on each testing criterion and considered three aspects: number of test cases necessary to satisfy the criterion, number of required elements, and infeasible elements generated by each criterion. Table 5 summarizes the cost information.

D. *Evaluation of effectiveness*

Effectiveness is the ability of a testing criteria to detect defects. To evaluate the effectiveness, the following equation was employed:

$$effectiveness = \frac{number\ of\ faults\ found}{number\ of\ faults\ injected} \quad (1)$$

Therefore, each faulty program was executed with the test set adequate for each testing criterion. In Table 10 is presented the percentage of defects revealed by each test set.

E. *Evaluation of the strength*

The strength of a criterion is related to the possibility of a test case set T_{C2} adequate to criterion $C2$ also be adequate for other criterion $C1$. The coverage achieved with the

[1] http://stoa.usp.br/silvanamm/files

application of T_{C2} in $C1$ demonstrates the difficulty of satisfaction a criterion. A high percentage of coverage indicates low strength and suggests that the criterion $C2$ can include the criterion $C1$. The results of the strength evaluation are provided in the next section.

5.3 Data collection

We executed all of our test cases on all of the faulty versions of the object programs. We applied sequential criteria and concurrent criteria to the runs of all faulty versions of the program to determine, for each test case if the fault was revealed. This allowed us collect necessary data to measure the dependent variables and compare the testing criteria.

5.4 Analysis of results and hypothesis testing

In this phase, the measurements collected during the experiment operation were analyzed and evaluated based on the principles of descriptive statistics. Descriptive statistics was used to organize and summarize data so that they could be better understood. Hypothesis testing was used to make inferences and draw conclusions about the population.

Hypothesis testing is performed to validate research and draw conclusions about the object of study. In this study, the object is the set of testing criteria. The sample is small and apparently the variations between populations are different, therefore we can not guarantee the normality of the data. Non-parametric tests were used during the hypothesis testing.

Statistical tests were used for the analysis of factors by groups (blocks) in two groups: (i) sequential testing criteria: all-nodes (AN), all-edges (AE), all-c-uses (ACU), and all-p-uses (APU); and (ii) concurrent testing criteria: all-p-nodes (ANP), all-w-nodes(AWP), all-s-edges(AES), all- comm-c-uses (ACCU), all-comm-p-uses (ACCP) and all-sync-uses (ASU).

In the following, the results are discussed and the research question Section 5.1.2 are answered, analyzing the defined hypotheses for the experiment.

5.4.1 Cost results

The hypothesis test showed significant cost differences between the criteria, mainly when the adequate test case set size and number of required elements were considered. Very similar cost values were found for the number of infeasible elements. In order to analyze the behavior of the sample selected for this study, the data were organized into bar and box plot charts for each factor analyzed.

Figures 7 and 8 show the cost related to the adequate test case set and number of required elements, in the side (a) the bar plot for the cost result and the side (b) the box plot for each criterion and the benchmark used. In the figures, on the x-axis, the black color represents the sequential criteria and blue color presents the concurrent criteria.

By means of the observation of these graphs, we can notice that the cost varies among the criteria. A classification can be established for the testing criteria in a decreasing cost order based on the results of statistic testing, as follows: all-edges, all-sync-uses, all-s-edges, all-nodes, all-comm-c-uses, all-p-uses, all-c-uses, all-p-nodes, all-comm-p-uses and all-w-nodes.

The Wilcoxon paired test (Wilcoxon 1945) was applied for the analysis of the null hypothesis $NH1$ related to the cost. This statistic testing was selected because of the type of comparison performed, in which the pairs of factors (criteria) of distinct groups (sequential

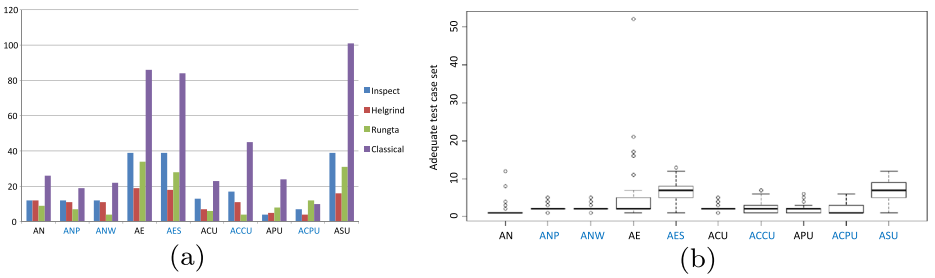


Fig. 7 Cost related to the size of the adequate test case set

or concurrent) was examined for the identification of differences among them. The null hypothesis $NH1$ is rejected if the median difference is not null or smaller than the confidence interval. The alternative hypothesis $AH1$ is accepted if the median difference is not zero and larger than an interval confidence established (in this case, larger than 0.05).

Table 7 show the test results of the Wilcoxon paired test between concurrent and sequential criteria considering the number of required elements generated by each criterion. The cost difference is not statistically significant for the pairs: ANP-AN, ANW-AN, ANP-AE, ANW-AE, AES-AE, ASU-AE, ANW-ACU. The concurrent criteria presented minor cost for the pairs: AES-AN, ASU-AN, ACCU-AN, ACPU-AN, ACCU-AE, ACPU-AE, ANP-ACU, ACCU-ACU, ACPU-ACU, ANP-APU, ANW-APU, AES-APU, ASU-APU, ACCU-APU, ACPU-APU. For the pairs AES-ACU and ASU-ACU, the concurrent criteria presented major cost than the sequential ones.

Table 8 shown the test results of the Wilcoxon paired test between concurrent and sequential criteria considering the adequate test set for each criterion. The concurrent criteria presented minor cost for the pairs ACCU-AN, ACPU-AN, ANP-ACU, ANW-ACU, ACCU-ACU, ACPU-ACU, ANW-APU, ACCU-APU e ACPU-APU. Only for the pairs ANP-AN, ANW-AN, ANP-AE, ANW-AE, AES-AE, ASU-AE, ANW-ACU was the difference not statistically significant. For the pairs AES-AN, ASU-AN, ACCU-AN, ACPU-AN, ACCU-AE, ACPU-AE, ANP-ACU, ACCU-ACU, ACPU-ACU, ANP-APU, ANW-APU, AES-APU, and ASU-APU, the concurrent criteria presented cost greater than the sequential criteria.

These results show evidence of the difference of the cost between the samples, concurrent criteria, and sequential ones. In addition, the analysis of Wilcoxon test indicates that, on average, the cost of concurrent criteria, considering the number of required elements, is higher than the cost spent on the sequential criteria. Therefore, it is possible to reject the

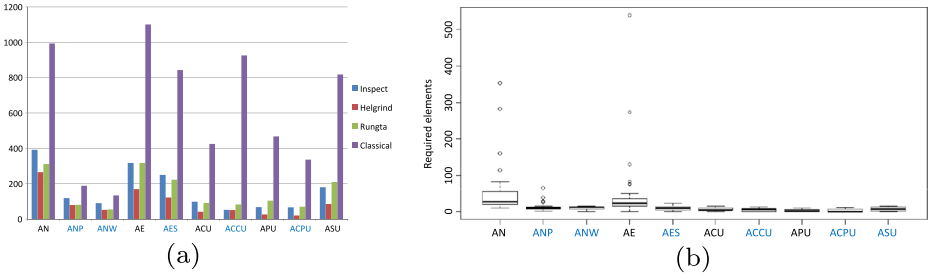


Fig. 8 Cost related to the number of required elements

Table 7 Results of cost comparison for Wilcoxon paired test considering the required elements

Factors	*P* value	W	Difference
ANP-AN	1.0000000	0	No
ANW-AN	1.0000000	0	No
AES-AN	0.9628906	8	Yes
ASU-AN	0.9628906	8	Yes
ACCU-AN	0.8085938	12	Yes
ACPU-AN	0.9843750	2	Yes
ANP-AE	1.0000000	0	No
ANW-AE	1.0000000	0	No
AES-AE	0.9961838	0	No
ASU-AE	0.9961838	0	No
ACCU-AE	0.9537543	6	Yes
ACPU-AE	0.9453125	5	Yes
ANP-ACU	0.99414062	3.0	Yes
ANW-ACU	0.99424309	0.0	No
AES-ACU	0.02490183	39.0	Yes
ASU-ACU	0.03303898	38.0	Yes
ACCU-ACU	0.58338419	16.5	Yes
ACPU-ACU	0.71093750	11.0	Yes
ANP-APU	0.9863281	5.0	Yes
ANW-APU	0.9896469	1.5	Yes
AES-APU	0.1313091	26.0	Yes
ASU-APU	0.2851562	28.0	Yes
ACCU-APU	0.5781250	17.0	Yes
ACPU-APU	0.6999103	8.0	Yes

null hypothesis $NH1$ and accept the alternative hypothesis $AH1$, concluding that, for this study, the cost of concurrent criteria is higher than the cost of the sequential criteria.

5.4.2 Effectiveness results

Based on the statistical results of effectiveness analysis, we can conclude that, on average, the criteria have a little difference in effectiveness to reveal defects. Figure 9 shows the effectiveness for each testing criterion. The bar plot in side (a) of the chart shows the average of effectiveness for each benchmark and criteria evaluated and the box plot in the side (b) illustrates the variability of the sample data.

The analysis of effectiveness also indicated some particular defect types were revealed only when criteria specific for concurrent programs were used. Table 9 shows the relationship of defect types and the criteria ability to reveal them.

Table 10 shows the analysis of the testing criteria. This table considers only testing criteria that generate required elements for the programs. The bold emphasis represents the average for each criterion (column) and each defect type (line). The table analysis shows that whenever it was possible, their application, the ACCU, ACPU, and ASU criteria were effective in revealing defects in different categories (except atomicity).

The results indicate that some injected faults can be revealed by different testing criteria while others cannot. This aspect is mainly related to the selected test cases. Some

Table 8 Results of cost comparison for Wilcoxon paired test considering the adequate test set

Factors	P-value	W	Difference
ANP-AN	0.967200154	0.0	No
ANW-AN	0.967200154	0.0	No
AES-AN	0.016407929	40.5	Yes
ASU-AN	0.008877961	28.0	Yes
ACCU-AN	0.375242710	12.0	Yes
ACPU-AN	0.624757290	9.0	Yes
ANP-AE	0.9961838	0	No
ANW-AE	0.9961838	0	No
AES-AE	0.9706091	0	No
ASU-AE	0.7673956	3	Yes
ACCU-AE	0.9910198	0	No
ACPU-AE	0.9915263	0	No
ANP-ACU	0.672639577	1.0	Yes
ANW-ACU	0.871580371	2.0	Yes
AES-ACU	0.008980239	28.0	Yes
ASU-ACU	0.003816221	45.0	Yes
ACCU-ACU	0.098733037	8.5	Yes
ACPU-ACU	0.293968373	9.5	Yes
ANP-APU	0.910143753	0.0	No
ANW-APU	0.929613613	1.0	Yes
AES-APU	0.013923975	41.0	Yes
ASU-APU	0.005756914	36.0	Yes
ACCU-APU	0.298056333	13.0	Yes
ACPU-APU	0.416742002	11.5	Yes

test cases that execute parts of the code with defects may generate a correct output masking the defects. The difference is that some coverage testing criteria are able to select test sets with high effectiveness due to required elements used to select those test sets, for instance, required elements related to communication and synchronization. As can be seen in Table 10, most defects are revealed by concurrent testing criteria. Another interesting result is that the sequential testing criteria all-nodes and all-edges present high effectiveness, indicating that covering elementary parts of the code is suitable to reveal some classes

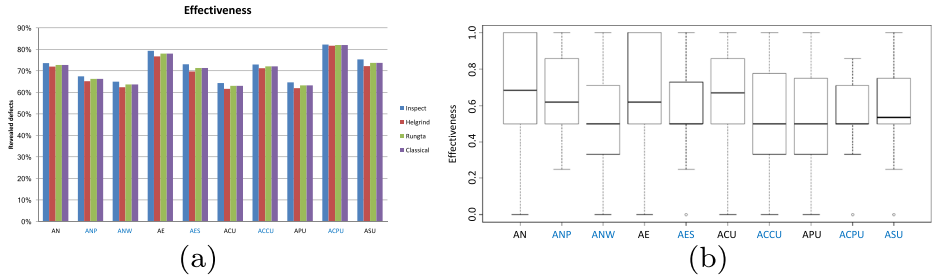


Fig. 9 Effectiveness of testing criteria

Table 9 Effectiveness of criteria for defect type

Defect type	Criteria that reveal defects
1. Deadlock	ACCU
2. Wrong lock	ACPU, ASU
3. Interference	ACCU, ACPU
4. Incorrect initialization	AE, AES, ASU, ACCU
5. Resource exhaustion	AN, ANP, ANW, AE, AES, ACU, APU, ASU, ACCU, ACPU
6. Blocking critical section	ACCU, ACPU
7. Atomicity	AN, AE
8. Sleep	AE, ACCU
9. Missing signals	AE, AES, ASU, ACPU
10. Missing a notify	AE, AES, ASU, ACPU

of defects, for instance, error of atomicity. These results are motivating and suggest the conduct of new empirical studies in which new dependent and independent variables must be defined and the defect types should be independent variable in order to measure the effect of the defects on the sample.

Friedman test (Friedman 1937) was used for the evaluation of the effectiveness of the two types of criteria. It is applied when the experimental tests involve blocks. According to the results of the overall analysis between the groups for the Friedman test considering the value of Chi-square Friedman, degrees of freedom (number of groups -1), and P-value for the sample the null hypothesis $NH2$ can be rejected and the alternative hypothesis $AH2$ should be accepted. Therefore, there is a significant difference in effectiveness between the criteria according to the 5% significance level.

The cost of concurrent criteria remains significantly greater than the sequential criteria due to interleaving explosion problem, even for a fixed set of inputs, there are a large number of concurrent runs that need to be tested, sharply increasing the number of required elements for the criteria and consequently the test cost. Therefore, the application of the specific testing criteria to concurrent programs with shared memory can be more effective and contribute to the test activity in revealing defects that are not discovered with the application

Table 10 Effectiveness for each testing criterion

Defect type	AN	ANP	ANW	AE	AES	ACU	APU	ASU	ACCU	ACPU	(AVG)
1. Deadlock	61%	56%	54%	64%	59%	53%	40%	59%	57%	44%	**55%**
2. Wrong lock	75%	57%	56%	82%	66%	54%	51%	64%	59%	52%	**62%**
3. Interference	31%	17%	15%	35%	24%	11%	17%	24%	24%	26%	**22%**
4. Incorrect initialization	77%	67%	63%	83%	80%	63%	57%	77%	63%	53%	**68%**
5. Resource exhaustion	69%	61%	61%	69%	61%	69%	46%	61%	54%	46%	**58%**
6. Blocking critical section	64%	58%	58%	69%	64%	56%	44%	67%	56%	42%	**58%**
7. Atomicity	50%	0%	0%	50%	0%	0%	0%	0%	0%	25%	**13%**
8. Sleep	60%	47%	47%	73%	60%	47%	47%	60%	67%	33%	**54%**
9. Missing signals	44%	44%	44%	50%	50%	37%	37%	50%	44%	37%	**48%**
10. Missing a notify	61%	61%	61%	69%	69%	54%	54%	69%	61%	46%	**61%**
Average (AVG)	**59%**	**47%**	**46%**	**64%**	**53%**	**44%**	**39%**	**53%**	**49%**	**40%**	**49%**

of only sequential testing criteria. The explosion of paths is a known problem and solutions from sequential programs (Wang et al. 2017) can be adapted for concurrent programs.

5.4.3 Strength results

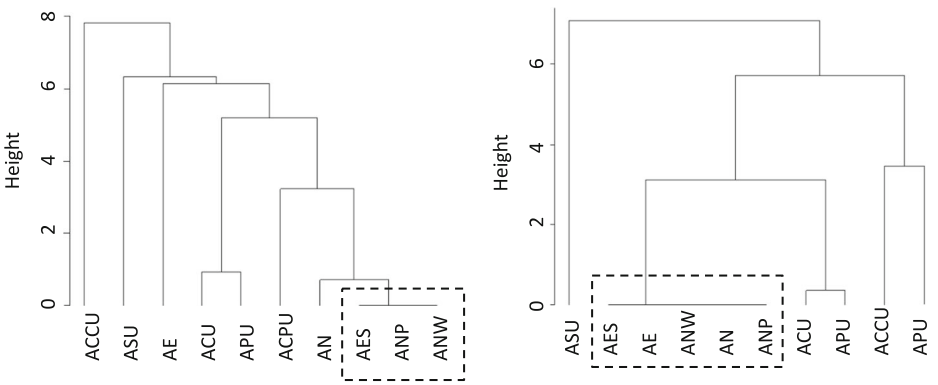
Dendrogram charts were designed based on the statistical method of cluster analysis and illustrate the results about strength, for the identification of possible inclusion relations among criteria. Vertical lines represent the distribution of a sample in relation to the other; the longer the line, the greater the difference between the samples. Samples that appear in the same edge connected by a horizontal line show higher similarity.

The cluster analysis method and dendrogram charts interpretation revealed the following relationships among the testing criteria: the all-nodes (AN) criterion includes all-w-nodes (ANW) and all-p-nodes (ANP) criteria, as expected, these criteria are derived from the all-nodes criterion; all-p-nodes (ANP), all-c-uses (ACU) and all-comm-c-uses (ACCU) criteria include only the all-w-nodes (AWN) criterion; the all-w-nodes (ANW) and all-comm-p-uses (ACPU) criteria have no inclusion relation with other criteria; the all-p-uses (APU) criterion includes the all-c-uses (ACU) criterion, the all-sync-uses (ASU) criterion includes all-w-nodes (ANW), all-p-nodes (ANP), and all-comm-p-uses (ACPU) criteria; the all-edges (AE) criterion includes all-s-edges (AES), all-nodes (AN), all-p-nodes (ANW) and all-p-nodes (ANP) criteria; all-s-edges (AES) criterion includes the all-w-nodes (ANW) and all-p-nodes (ANP) criteria.

Figure 10a shows the all-sync-uses (ASU) criterion has the highest disparity of distribution in relation to the other criteria, because it has the highest and most isolated vertical line. The all-comm-p-uses (ACPU) and all-comm-c-uses (ACCU) criteria, and all-p-uses (APU) and all-c-uses (ACU) criteria, show a significant difference in the distribution of their samples.

According to Fig. 10b, no vertical line separates AES, ANP, and ANW criteria; they are in the same horizontal line, which indicates there are no significant differences among them. Therefore, all-edges (AE) criterion includes all-s-edges (AES), all-nodes (AN), all-p-nodes (ANW), and all-p-nodes (ANP) criteria, i.e., they are inclusive.

Based on these results, the null hypothesis $NH3$ can be rejected and the alternative hypothesis $AH3$ is accepted, indicating that the criteria can be complementary.



(a) Strength of All-edges-s criterion. (b) Strength of All-edges criterion.

Fig. 10 Dendrogram charts of strength factor

Table 11 Results of hypothesis testing for the three hypotheses established

Hypothesis	Definition	Results
NH1	The application cost is the same for all sequential and concurrent criteria	Rejected
AH1	The application cost is different for sequential and concurrent criteria	Accepted
NH2	Effectiveness is the same for all sequential and concurrent criteria	Rejected
AH2	Effectiveness is different for sequential and concurrent criteria	Accepted
NH3	There is not one criterion that subsumes another	Rejected
AH3	There is at least one criterion that subsumes another	Accepted

The strength factor was evaluated through the statistical method of cluster analysis for the identification of inclusion relations among the testing criteria. The analysis of the dendrograms in Fig. 10a and b shows there exist inclusion relations between the criteria, therefore the null hypothesis $NH3$ should be rejected and the alternative hypothesis $AH3$ should be accepted.

Table 11 summarizes the results of all hypothesis tests presented on the sections above. In general, testing criteria has significant differences of cost, and the concurrent criteria tend to be less expensive than the sequential ones. Regarding the effectiveness factor, the testing criteria presented small differences. Results of the strength factor and the cluster analysis showed the possibility of inclusion relations among the criteria.

6 Discussion

A good test strategy needs to be highly effective and demands low cost. These measures are strongly related. Thus, a testing criterion with high effectiveness can be prohibitive to apply if the cost is high (Brito et al. 2013).

Based on the results obtained with the experimental evaluation, we proposed a testing strategy to apply these testing criteria. The strategy is based on the fundamentals from a sequential program and allows covering different aspects for the test. There are four possible ways to apply the strategy, as defined as follows:

1. Starts the application from all-sync-uses criterion and cover in sequence: all-edges, all-edges-s, all-w-nodes, and all-p-nodes criteria or starts to all-syn-uses and cover all-nodes, all-p-nodes, and all-w-nodes.
2. Starts the application from the all-p-use criterion and cover all-nodes, all-p-nodes, and all-w-nodes or just cover the all-communication-p-use from the all-p-use.
3. Starts the application from the all-c-use criterion and cover all-p-nodes and all-w-nodes criteria, or just cover the all-communication-p-use from all-c-use.
4. Starts the application from all-edges and cover all-nodes, all-p-nodes, and all-w-nodes or starts from all-edges and cover all-edges-s, all-p-nodes, and all-w-nodes.

Figure 11 shows the possible hierarchy of application sequence to structural testing criteria for multithreaded programs. The tester can choose the best way to start considering the information that is required to cover. We suggest the strategy 1 to cover communication and synchronization between threads; 2 and 3 to cover variables usage; and 4 to cover sequential control flow. These strategies can be used together to achieve a high coverage of the code improving the quality of the test activity.

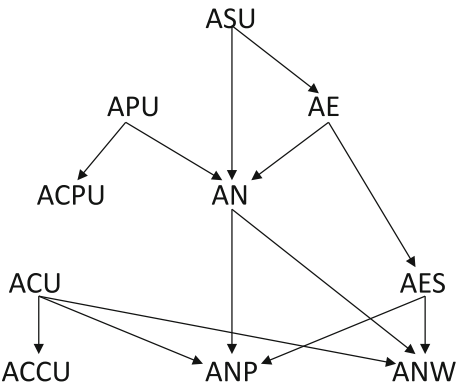


Fig. 11 Application strategy scheme

To choose the adequate criterion to apply, it is necessary to consider cost and effectiveness of the criterion and the restrictions and features of the testing project (Vegas and Basili 2005).

In a hypothetical practical scenario where the test must be applied in a critical system, in which faults result in loss of lives or money, the most important factor to the testing project is the effectiveness to reveal faults. In this case, the effectiveness of each testing criterion should guide the choice of the testing strategy and a possible solution is to apply the strategy 1 or 4 as defined above. On the other hand, consider a system in which the faults do not lead to serious damage and the testing project has limited resources (as time and money). In this case, strategy 2 or 3 could be applied and then, if more resource will be allocated to the project, more effective testing criteria may be added in an incremental way.

The choice of the best testing criterion is still a challenge and some studies have investigated the benefits of the different testing techniques and their relation to software project features (da Costa Araújo et al. 2016; Vos et al. 2012; Society et al. 2014). Our results, while still initial, offer possible test strategies that benefit testers of concurrent and parallel programs and collaborate to improve the software quality in this software domain.

6.1 Threats to validity

Some possible threats to validity have been identified in this study, however, we believe that they do not affect drastically the results and do not compromise the validity of these research contributions.

6.1.1 Conclusion validity

The conclusion validity corresponds to the obtaining of correct conclusions about the theory analyzed. Experimental conclusions should be influenced by the statistical test chosen. The statistical tests were selected based on the normality analysis and characteristics of the data collected, and following the fundamentals of experimental statistics. Specialists in the area were consulted during the interpretation and data analysis, therefore, we believe that these threats to validity have been avoided.

6.1.2 Internal validity

Internal validity is related to the conclusions about the causes and their effects in the study. The differences between the factors analyzed (cost, effectiveness, and strength) may result

from other factors, such as validity of the required elements generated by the testing tool, and not from the cause analyzed (type of criteria). However, the testing tool was developed by our research group and the validation of the tool included to analyze if the tool generates the correct outputs. Thus, we can consider this is not a threat of our experiment.

6.1.3 Construct validity

Construct validity threats concern the relationship between theory and observation. In this study, it is related to the knowledge about the programs used in the experiment and faults inserted that may influence the application of the testing criterion. To avoid this threat, the order of criteria application was different for each program, the faults were inserted by others and the programs were extracted from the literature.

6.1.4 External validity

External validity refers to the possibility of generalizing the results of the experiment for another context. This study was conducted to enable replication and comparison of new and different approaches to concurrent program testing. The programs that compose the experiment contemplates different features of concurrent programs generally used in the research. However, it is possible that these programs are not representative of all multithreaded programs and our results may not properly generalize the area in terms of scale and complexity of industrial applications. To avoid this threat, the programs that compose the experiment contemplate different features of concurrent programs.

7 Concluding remarks and future work

Software testing is one of the main quality-assurance activities. To apply this activity efficiently, support mechanisms are essential, such as testing tools and test data selection criteria. In the context of sequential programs, a vast number of tools and testing criteria have been proposed and several experimental studies have been conducted in order to evaluate the application and effectiveness of those testing criteria. In the context of multithreaded programs, we can find some proposals of tools and testing approaches, however, the empirical evaluation is incipient.

This paper contributes to this scenario, presenting a set of structural testing criteria for multithreaded programs that supports the validation of the control, data, communication, and synchronization flow of these programs. To support the application of these criteria, a testing tool, ValiPthread, is presented.

The main contribution of this paper is to present a complete experimental study to evaluate our structural testing criteria in order to show the benefits and limitations of them. This study is innovative because it applies the concepts from experimental software engineering (ESE) (Wohlin et al. 2000) to develop the empirical evaluation. Also, all material collected and generated during the experiment, including programs, fault programs, testing tools, and results are available on-line[1] allowing further studies and comparisons with others testing approaches.

Experimental results indicate that the concurrent testing criteria present a higher application cost in comparison to sequential testing criteria with higher effectiveness in revealing faults.

In relation to cost, we analyzed two metrics: (1) size of test set necessary to cover the required elements of each testing criterion and (2) amount of required elements of each

testing criterion. These metrics can properly measure the effort for applying coverage-testing criteria. The results demonstrate that the cost varies among the testing criteria and the following coverage criteria present higher cost: all-edges (AE), all-sync-uses (ASU), and all-s-edges (AES). However, when the effectiveness is analyzed, we observe that the all-sync-uses (ASU) criterion presents higher effectiveness and it can reveal faults in most errors category (except atomicity). In this case, we have a clear trade-off in relation to cost versus effectiveness. Results also indicate that the all-comm-c-uses (ACCU) criterion is a good alternative because it presents lower cost (compared to other coverage criteria) and higher effectiveness. We decided to compare concurrent and sequential test criteria to evaluate the benefits of defining coverage criteria for aspecific application domain. These results indicate that specific coverage criteria (or concurrent testing criteria in our context) showed advantages. For instance, some kinds of faults are only revealed by the testing criteria proposed for multithreaded programs: deadlock, wrong lock, interference, and blocking critical section. These results motivate the definition and analysis of criteria specific to concurrent programs. In some cases, these criteria can improve the effectiveness of revealing concurrent faults.

Based on these results, we propose a testing strategy to apply these testing criteria. The strategy allows us to cover different aspects of the multithreaded program, including testing specific features of the concurrent program, testing the data flow into the program, or testing the sequential flow of the program.

More work is necessary to evaluate the relevance of these testing criteria in relation to other testing approaches for mulithreaded programs. Our research group has developed a variety of testing criteria for concurrent languages Souza et al. (2008a, b, 2012a, 2015a), different testing techniques (Giacometti et al. 2002; Silva et al. 2012) and different application contexts (Brito et al. 2015; Souza et al. 2007) that can be compared in order to provide useful information for the research community about the effectiveness of these approaches to guide the selection of appropriate testing techniques for a specific context and estimate their applicability.

Currently, we are conducting work in the direction of defining a knowledge base useful in conducting research in this area. We gathered and classified proposed tools that implement testing techniques for concurrent programs (Melo et al. 2015) to provide a useful catalog that helps testing practitioners and researchers in the testing technique selection procedure. In order to improve the conduct and comparison of empirical studies, we proposed a preliminary design for empirical studies in this field (Melo et al. 2016). A set of benchmarks of programs for different languages is proposed aiming to provide a repository of more complex and real applications for concurrent software testing (Dourado et al. 2016). We aim to repeat this work using the design of experiment, real industrial application, and other resources provided by our benchmark to validate the research and to study how these criteria work for more complex applications when compared with other related approaches in this research area.

Acknowledgements The authors acknowledge the State of São Paulo Research Foundation - FAPESP, for the financial support (under processes no. 2010/04042-1, 2013/05046-9, 2013/01818-7 and 2015/23653-5) provided to this research.

References

Badlaney, J., Ghatol, R., & Jadhwani, R. (2006). *An introduction to data-flow testing*. Tech. Rep. 22, North Carolina State University, Raleigh.

Basili, V.R. (1996). The role of experimentation in software engineering: past, current, and future. In *ICSE* (pp. 442–449).

Bradbury, J.S., & Jalbert, K. (2009). Defining a catalog of programming anti-patterns for concurrent java. In *Proceedings of SPAQu'09* (pp. 6–11).

Brito, M.A.S., do Rocio Senger de Souza, S., & de Souza, P.S.L. (2013). An empirical evaluation of the cost and effectiveness of structural testing criteria for concurrent programs. In *ICCS* (Vol. 18, pp. 250–259). Elsevier, Procedia.

Brito, M.A.S., Santos, M., Souza, P.S.L., & Souza, S.R.S. (2015). Integration testing criteria for mobile robotic systems. In *The 27th international conference on software engineering and knowledge engineering, SEKE 2015, July 6–8, 2015* (pp. 182–187). Pittsburgh: Wyndham Pittsburgh University Center.

Carver, R.H., & Lei, Y. (2010). Distributed reachability testing of concurrent programs. *Concurrency and Computation: Practice and Experience*, *22*(18), 2445–2466.

Carver, R.H., & Tai, K.C. (1991). Replay and testing for concurrent programs. *IEEE Software*, *8*(2), 66–74.

Chung, C.M., Shih, T.K., Wang, Y.H., Lin, W.C., & Kou, Y.F. (1996). Task decomposition testing and metrics for concurrent programs. In *Fifth international symposium on software reliability engineering* (pp. 122–130).

Cordeiro, L., & Fischer, B. (2011). Verifying multi-threaded software using smt-based context-bounded model checking. In *33rd international conference on software engineering, ICSE* (pp. 331–340). New York: ACM.

da Costa Araújo, I., da Silva, W.O., de Sousa Nunes, J.B., & Neto, F.O. (2016). Arrestt: a framework to create reproducible experiments to evaluate software testing techniques. In *Proceedings of the 1st Brazilian symposium on systematic and automated software testing, SAST* (pp. 1:1–1:10). New York: ACM. doi:10.1145/2993288.2993303.

Damodaran-Kamal, S.K., & Francioni, J.M. (1993). Nondeterminacy: testing and debugging in message passing parallel programs. In *3rd ACM/ONR workshop on parallel and distributed debugging* (pp. 118–128). ACM.

de Oliveira Neto, F.G., Torkar, R., & Machado, P.D.L. (2015). An initiative to improve reproducibility and empirical evaluation of software testing techniques. In *Proceedings of the 37th international conference on software engineering - Volume 2* ICSE'15 (pp. 575–578). Piscataway: IEEE Press.

Denaro, G., Pezzè, M., & Vivanti, M. (2013). Quantifying the complexity of dataflow testing. In *8th international workshop on automation of software test, AST 2013, May 18–19* (pp. 132–138). San Francisco.

Dourado, G.G.M., de Souza, P.S.L., Prado, R.R., Batista, R.N., Souza, S.R.S., Estrella, J.C., Bruschi, S.M., & Lourenço, J. (2016). A suite of Java message-passing benchmarks to support the validation of testing models, criteria and tools. In *International conference on computational science 2016, ICCS 2016, 6–8 June, 2016*, (pp. 2226–2230). San Diego, California.

Edelstein, O., Farchi, E., Golden, E., Nir, Y., Ratsaby, G., & Ur, S. (2002). Contest: a users perspective. In *5th international conference on achieving quality in software*. Venezia.

Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., & Ur, S. (2003). Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, *15*(3–5), 485–499.

Farchi, E., Nir, Y., & Ur, S. (2003). Concurrent bug patterns and how to test them. In *17th international parallel and distributed processing symposium (IPDPS 2003) - workshop on parallel and distributed systems: testing and debugging* (pp. 286–293). Nice: IEEE Computer Society.

Foreman, L.M., & Zweben, S.H. (1993). A study of the effectiveness of control and data flow testing strategies. *Journal of Systems and Software*, *21*(3), 215–228.

Frankl, F.G., & Weyuker, E.J. (1986). Data flow testing in the presence of unexecutable paths. In *Workshop on software testing* (pp. 4–13). Banff.

Friedman, M. (1937). The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, *32*(200), 675–701.

Giacometti, C., Souza, S.R.S., & Souza, P.S.L. (2002). Teste de mutação para a validação de aplicações concorrentes usando PVM. REIC. Revista Eletrônica de Iniciação Científica, v. II, n. III.

Grama, A., Karypis, G., Kumar, V., & Gupta, A. (2003). *Introduction to parallel computing, 2nd Edn*. Reading: Addison Wesley.

Hong, S., Staats, M., Ahn, J., Kim, M., & Rothermel, G. (2013). The impact of concurrent coverage metrics on testing effectiveness. In *2013 IEEE 6th international conference on software testing, verification and validation* (pp. 232–241).

Hong, S., Staats, M., Ahn, J., Kim, M., & Rothermel, G. (2015). Are concurrency coverage metrics effective for testing: a comprehensive empirical investigation. *Software Testing Verification and Reliability*, *25*(4), 334–370.

Höst, M., Regnell, B., & Wohlin, C. (2000). Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, *5*(3), 201–214.

Hutchins, M., Foster, H., Goradia, T., & Ostrand, T. (1994). Experiments of the effectiveness of dataflow and control flow based test adequacy criteria. In *16th international conference on software engineering, ICSE '94* (pp. 191–200). Los Alamitos: IEEE.

Jalbert, N., & Sen, K. (2010). A trace simplification technique for effective debugging of concurrent programs. In *18th ACM SIGSOFT international symposium on foundations of software engineering, FSE '10* (pp. 57–66). New York: ACM.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, *21*(7), 558–565.

Lei, Y., & Carver, R.H. (2006). Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, *32*(6), 382–403.

Li, N., Praphamontripong, U., & Offutt, J. (2009). An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *Second international conference on software testing verification and validation, Denver, Colorado, USA, April 1-4, 2009, Workshops Proceedings* (pp. 220–229). Los Alamitos: IEEE Computer Society.

Lu, S., Park, S., Seo, E., & Zhou, Y. (2008). Learning from mistakes: a comprehensive study on real-world concurrency bug characteristics. *SIGOPS Operating Systems Review*, *42*, 329–339.

Lu, S., Park, S., & Zhou, Y. (2012). Finding atomicity-violation bugs through unserializable interleaving testing. *IEEE Transactions on Software Engineering*, *38*(4), 844–860.

Mathur, A.P., & Wong, E.W. (1993). An empirical comparison of mutation and data flow based test adequacy criteria. *Journal of Software Testing, Verification, and Reliability*, *4*(1), 9–31.

Mathur, A.P., & Wong, W.E. (1994). An empirical comparison of data flow and mutation-based test adequacy criteria. *The Journal of Software Testing, Verification and Reliability*, *4*(1), 9–31.

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, *2*, 308–320.

Melo, S.M., Souza, S.R.S., & L, S.P.S. (2012). Structural testing for multithreaded programs: an experimental evaluation of the cost, strength and effectiveness. In: *24th international conference on software engineering & knowledge engineering (SEKE'2012), July, 2012* (Vol. 1–3, pp. 476–479). San Francisco Bay.

Melo, S.M., Souza, S.R.S., Silva, R.A., & Souza, P.S.L. (2015). Concurrent software testing in practice: a catalog of tools. In *Proceedings of the 6th international workshop on automating test case design, selection and evaluation, A-TEST 2015* (pp. 31–40). New York: ACM.

Melo, S.M., Souza, P.S.L., & Souza, S.R.S. (2016). Towards an empirical study design for concurrent software testing. In *Fourth international workshop on software engineering for high performance computing in computational science and engineering, SC '16* (p. 49). Salt Lake City: IEEE Press.

Mühlenfeld, A., & Wotawa, F. (2007). Fault detection in multi-threaded C++ server applications. *Electronic Notes in Theoretical Computer Science*, *174*(9), 5–22.

Musuvathi, M., Qadeer, S., & Ball, T. (2007). *Chess: a systematic testing tool for concurrent software*. Tech. Rep. MSR-TR-2007-149, Microsoft Research.

Ntafos, S.C. (1988). A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, *14*(6), 868–873.

Offutt, A.J., Pan, J., Tewary, K., & Zhang, T. (1996). An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, *26*(2), 165–176.

Rapps, S., & Weyuker, E.J. (1985). Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, *11*(4), 367–375. doi:10.1109/TSE.1985.232226.

Rungta, N., & Mercer, E.G. (2009). A meta heuristic for effectively detecting concurrency errors. *Lecture Notes in Computer Science LNCS*, *5394*, 23–37.

Sarmanho, F.S., Souza, P.S., Souza, S.R., & Simão, A.S. (2008). Structural testing for semaphore-based multithread programs. In *Proceedings of the 8th international conference on computational science, Part I, ICCS '08* (pp. 337–346). Berlin: Springer.

Silva, R.A., do Rocio Senger de Souza, S., & de Souza, P.S.L. (2012). Mutation operators for concurrent programs in MPI. In *13th Latin American test workshop, LATW 2012, April 10–13, 2012* (pp. 1–6). Quito.

Simao, A.S., Vincenzi, A.M.R., Maldonado, J.C., & Santana, A.C.L. (2003). A language for the description of program instrumentation and the automatic generation of instrumenters. *CLEI Electronic Journal*, **6**(1).

Society, I.C., Bourque, P., & Fairley, R.E. (2014). *Guide to the software engineering body of knowledge (SWEBOK(R)): Version 3.0, 3rd Edn*. Los Alamitos: IEEE Computer Society Press.

Souza, S., Vergilio, S., Souza Pao, A.S., Bliscosque, T., Lima, A., & Hausen, A. (2005). Valipar: a testing tool for message-passing parallel programs. In *International conference on software knowledge and software engineering (SEKE05)* (pp. 386–391). Taipei-Taiwan.

Souza, S., Sugeta, T., Fabbri, S., Masiero, P., & Maldonado, J. (2007). Coverage testing criteria for statecharts specifications validation. *Software, Testing, Verification and Reliability* Submitted.

Souza, P.L., Sawabe, E.T., Simão, A.S., Vergilio, S.R., & Souza, S.R.S. (2008a). ValiPVM—a graphical tool for structural testing of PVM programs. In *Proceedings of the 15th European PVM/MPI users' group meeting on recent advances in parallel virtual machine and message passing interface* (pp. 257–264). Berlin: Springer.

Souza, S.R.S., Vergilio, S.R., Souza, P.S.L., Simão, A.S., & Hausen, A.C. (2008b). Structural testing criteria for message-passing parallel programs. *Concurrency and Computation: Practice and Experience*, *20*, 1893–1916.

Souza, S.R.S., Prado, M.P., Barbosa, E.F., & Maldonado, J.C. (2012b). An experimental study to evaluate the impact of the programming paradigm in the testing activity. *CLEI Electronic Journal (Online)*, *15*(1), 4–4.

Souza, P.S.L., Souza, S.R.S., & Zaluska, E. (2012a). Structural testing for message-passing concurrent programs: an extended test model. *Concurrency and Computation: Practice and Experience*, *26*(1), 21–50.

Souza, P.S., Souza, S.S., Rocha, M.G., Prado, R.R., & Batista, R.N. (2013). Data flow testing in concurrent programs with message passing and shared memory paradigms. *Procedia Computer Science*, *18*, 149–158.

Souza, S.R.S., Souza, P.S.L., Brito, M.A.S., da Silva Simão, A., & Zaluska, E. (2015a). Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs. *Software Testing, Verification and Reliability*, *25*(3), 310–332.

Souza, S.R.S., Souza, P.S.L., Brito, M.A.S., Simao, A.S., & Zaluska, E.J. (2015b). Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs. *Software Testing, Verification and Reliability*, *25*(3), 310–332.

Takahashi, J., Kojima, H., & Furukawa, Z. (2008). Coverage based testing for concurrent software. In *28th international conference on distributed computing systems workshops. ICDCS' 08* (pp. 533–538). Beijing: IEEE.

Tanenbaum, A.S. (1995). *Distributed operating systems*. Upper Saddle River: Prentice-Hall, Inc.

Taylor, R.N., Levine, D.L., & Kelly, C. (1992). Structural testing of concurrent programs. *IEEE Transaction Software Engineering*, *18*(3), 206–215.

Valgrind-Developers (2014). Valgrind-3.6.1. http://valgrind.org/. Accessed 16 December 2014.

Vegas, S., & Basili, V. (2005). A characterisation schema for software testing techniques. *Empirical Software Engineering*, *10*(4), 437–466. doi:10.1007/s10664-005-3862-1.

Vergilio, S.R., Souza, S.R.S., & Souza, P.S.L. (2005). Coverage testing criteria for message-passing parallel programs. In *LATW2005 - 6th IEEE Latin-American test workshop* (pp. 161–166). Salvador.

Vergilio, S.R., Maldonado, J.C., & Jino, M. (2006). Infeasible paths in the context of data flow based testing criteria: identification, classification and prediction. *Journal of the Brazilian Computer Society*, *12*(1), 73–88.

Vos, T.E.J., Marín, B., Escalona, M.J., & Marchetto, A. (2012). A methodological framework for evaluating software testing techniques and tools. In *2012 12th international conference on quality software* (pp. 230–239). doi:10.1109/QSIC.2012.16.

Wang, C., Chaudhuri, S., Gupta, A., & Yang, Y. (2009). Symbolic pruning of concurrent program executions. In *7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, ESEC/FSE '09* (pp. 23–32). New York: ACM.

Wang, H., Liu, T., Guan, X., Shen, C., Zheng, Q., & Yang, Z. (2017). Dependence guided symbolic execution. *IEEE Transactions on Software Engineering*, *43*(3), 252–271.

Wesonga, S., Mercer, E.G., & Rungta, N. (2011). Guided test visualization: making sense of errors in concurrent programs. In *26th IEEE/ACM international conference on automated software engineering, ASE* (pp. 624–627). Washington, DC: IEEE.

Weyuker, E.J. (1990). The cost of data flow testing: an empirical study. *IEEE Transactions on Software Engineering*, *16*(2), 121–128.

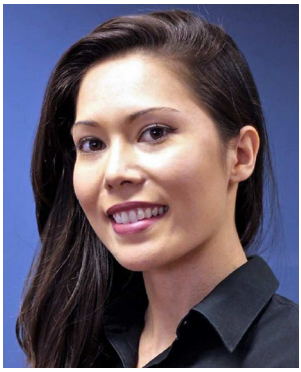
Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics*, *1*, 80–83.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., & Wesslén, A. (2000). *Experimentation in software engineering: an introduction*. Norwell: Kluwer Academic Publishers.

Wong, W.E., Lei, Y., & Ma, X. (2005). Effective generation of test sequences for structural testing of concurrent programs. In *10th IEEE international conference on engineering of complex computer systems (ICECCS'05)* (pp. 539–548). Los Alamitos: IEEE. doi:10.1109/ICECCS.2005.37.

Xiao, X., Xie, T., Tillmann, N., & Halleux, J. (2011). Precise identification of problems for structural test generation. In *Proceedings of the 33rd international conference on software engineering (ICSE, 2011)* (pp. 611–620). Honolulu: IEEE.

Yang, Y. (2014). Inspect: a framework for dynamic verification of multithreaded C programs. http://www.cs.utah.edu/yuyang/inspect/. Accessed 16 December 2014.
Yang, R.D., & Chung, C.G. (1992). Path analysis testing of concurrent programs. *Information and Software Technology*, *34*, 43–56.
Yang, C.S., & Pollock, L.L. (1997). The challenges in automated testing of multithreaded programs. In *14th International conference on testing computer software* (pp. 157–166).
Yang, C.S.D., & Pollock, L.L. (2003). All-uses testing of shared memory parallel programs. *Software Testing, Verification and Reliability Journal*, *13*, 3–24.
Yang, C.S.D., Souter, A.L., & Pollock, L.L. (1998). All-du-path coverage for parallel programs. *SIGSOFT Software Engineering Notes*, *23*, 153–162.
Yang, Y., Chen, X., Gopalakrishnan, G., & Kirby, R.M. (2008). Efficient stateful dynamic partial order reduction. In K. Havelund, R. Majumdar, J. Palsberg (Eds.), *SPIN, Lecture notes in computer science* (Vol. 5156, pp. 288–305). Springer.
Yastrebenetsky, P., & Trakhtenbrot, M. (2011). Analysis of applicability for synchronization complexity metric. In *18th IEEE international conference and workshops on engineering of computer-based systems, ECBS '11* (pp. 24–33). Washington, DC: IEEE.
Zhu, H. (1996). A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering*, *22*, 248–255.



Silvana Morita Melo holds a B.Sc. degree in Information Systems from Federal University of Mato Grosso do Sul - UFMS (2009) and a M.Sc. in Computer Science and Computational Mathematics from University of Sao Paulo - USP (2013). She is currently a Ph.D. student in the Institute of Mathematics and Computer Science (ICMC/USP) and a visiting student researcher at the University of Alabama (UA). Her research interests include software testing, concurrent programming, concurrent software testing, and empirical software engineering.

Simone do Rocio Senger de Souza holds a joint MSc (1996) and PhD (2000) degree in Computer Science from the University of Sao Paulo (USP), Brazil. From 2010 to 2011 she was a visiting scientist at the University of Southampton, England. She is an associate professor of software engineering at the Institute of Mathematics Science and Computer at the University of Sao Paulo (ICMC/USP, Brazil) since 2005. In the past, she was a lecturer at the Informatics Department at the State University of Ponta Grossa (UEPG), Parana, Brazil from 1991 to 2005. Currently, she has conducted research in concurrent software testing, structural testing, mutation testing, and software engineering experimentation.

Felipe Santos Sarmanho has bachelor in Computer Science by the Catholic University of Pernambuco - UNICAP (2005) and M.Sc. from the University of Sao Paulo - USP (2009). He has experience in the area of Computer Science, with knowledge in Computer Systems Architecture, acting mainly as an infrastructure support analyst. Currently, he manages Data Center with virtualization, basic software support, and backup routine management. He has already worked in the area of computational complexity of vector quantization and in the development of corporate software and software for literacy of children with special needs.

Paulo Sergio Lopes de Souza holds a joint MSc (1996) and PhD (2000) degree in Computer Science from the University of Sao Paulo (USP), Brazil. From 2010 to 2011 he was a visiting scientist at the University of Southampton, England. He is an associate professor of distributed systems at the Institute of Mathematics Science and Computer at the University of Sao Paulo (ICMC/USP, Brazil) since 2005. In the past, he was a lecturer at the Informatics Department at the State University of Ponta Grossa (UEPG), Parana, Brazil from 1991 to 2005. Currently, he has conducted research in distributed systems, parallel computing, development of high-performance applications, software testing, and open educational resources applied to computer education.