Structural Testing for Semaphore-Based Multithread Programs

Conference Paper · June 2008

DOI: 10.1007/978-3-540-69384-0_39 · Source: DBLP

CITATIONS
20

READS
5,402

4 authors:

Felipe Santos Sarmanho
Agencia Nacional de Aviação Civil, Brasilia, Brasil
2 PUBLICATIONS **23** CITATIONS
SEE PROFILE

Paulo Sergio Lopes de Souza
University of São Paulo
84 PUBLICATIONS **402** CITATIONS
SEE PROFILE

Simone Do Rocio Senger de Souza
University of São Paulo
109 PUBLICATIONS **712** CITATIONS
SEE PROFILE

Adenilso da Silva Simão
University of São Paulo
117 PUBLICATIONS **996** CITATIONS
SEE PROFILE

Structural Testing for Semaphore-Based Multithread Programs*

Felipe S. Sarmanho, Paulo S.L. Souza,
Simone R.S. Souza, and Adenilso S. Simão

Universidade de São Paulo, ICMC, São Carlos - SP, 668, Brazil
{sarmanho,pssouza,srocio,adenilso}@icmc.usp.br

Abstract. This paper presents structural testing criteria for validation of semaphore-based multithread programs exploring control, data, communication and synchronization information. A post-mortem method based on timestamps is defined to determine the implicit communication among threads using shared variables. The applicability of the coverage testing criteria is illustrated by a case study.

Keywords: software testing, multithread programs, testing criteria.

1 Introduction

Concurrent programming is important to reduce the execution time in several application domains, such as image processing and simulations. A concurrent program is a group of processes (or threads) that execute simultaneously and work together to perform a task. These threads access a common addressing space and interact through memory (using shared variables). The most common method to develop multithread programs is to use thread libraries, like PThreads (POSIX Threads).

Concurrent program testing is not trivial. Features like synchronization, inter-thread communication and non-determinism make this activity complex [1]. Multiple executions of a concurrent program with the same input may present different results due to different synchronization and communication sequences. Petascale systems also add more factors to this scenario, making it even worse [2].

Structural testing is a test technique that use source code information to guide the testing activity. Coverage criteria are defined to apply structural testing. A testing criterion is a predicate to be satisfied by a set of test cases and can be used as a guide for the test data generation. It is also a good heuristic to indicate defects on programs and thus to improve their quality. This activity is composed of: (1) *static analysis* to obtain the necessary data about the source code, and usually obtaining a Control Flow Graph (CFP) [3]; (2) *determining required elements* for the coverage criterion chosen; and (3) *analyzing the coverage* reached in source code by test cases, based on coverage criterion.

* This work is supported by CNPq.

M. Bubak et al. (Eds.): ICCS 2008, Part I, LNCS 5101, pp. 337–346, 2008.

In the literature there are some works that address testing of concurrent programs [4, 5, 6, 7, 8, 9]. Most of these works propose a test model to represent the concurrent program and to support the testing application.

The Concurrency States Graph (*CG*) is a CFG extension proposed in [4], in which nodes represent concurrency states while the edges represent the actions required for transition between these states. That work considers concurrent languages with explicit synchronization using rendezvous-style mechanisms, such as Ada and CSP. It presents coverage criteria adapted for the *CG*; however, its usage is limited in the practice by the state space explosion problem.

PPFG (Parallel Program Flow Graph) is a graph where was inserted the concept of synchronization node to the *CFG* [5, 10]. In the *PPFG* each process that composes the program has its own *CFG*. The synchronization nodes are then connected based on possible synchronizations. This model was proposed to adapt the all-du-path criterion to concurrent programs.

PCFG (Parallel Control Flow Graph) also adapts the *CFG* for the context of parallel programs in message passing environments [7]. The *PCFG* includes the concept of synchronization nodes that are used to represent the send and receive primitives. The concept of variables was extended, to consider the concept of communicational use (s-use). Coverage criteria were also proposed in [7], based on models of control and data flow for message passing programs.

Lei and Carver propose an approach to reachability testing. Reachability testing is a combination of deterministic and non-deterministic execution, where the information and the required elements are generated on-the-fly, without static analysis [6]. This proposal guarantees all feasible synchronization sequences will be exercised at least once. The lack of static analysis means it cannot say how many executions are required. This causes the state space explosion problem. In recent works, Lei et. al [9] presents a combinatorial approach, called t-way, to reduce the number of synchronization sequences to be executed.

These related works bring relevant improvements for concurrent program testing. However, few works are found that investigate the application of the testing coverage criteria and supporting tools in the context of multithreading programs. For these programs, new aspects need to be considered. For instance, data flow information must consider that an association between one variable definition and its use can occur in different threads. The implicit inter-thread communication that occurs through shared memory makes complex the test activity. The investigation of these challenges it is not a trivial task and presents many difficulties. To overcome these difficulties, we present a family of structural testing criteria for semaphore-based multithread programs and a new test model for the support to the criteria. This model includes important features, such as: synchronization, communication, parallelism and concurrency. These data are collected using static and dynamic analyses. Information about communication is obtained after the execution of an instrumented version of the program, using a post-mortem methodology. This methodology has been adapted from Lei and Carver work [6]. Testing criteria were defined to exploit the control and data flows of these programs, considering their sequential and parallel aspects. The

main contribution of the testing criteria proposed in this paper is to provide a coverage measure that can be used for evaluating the progress of the testing activity. This is important to evaluate the quality of test cases, as well as, to consider that a program has been tested enough. It is important to point out that the objective this work is not to debug concurrent programs which already have an error revealed.

2 Model Test for Shared Memory Programs

Let $MT = \{t^0, t^1, ..., t^{n-1}\}$ be a multithread program composed of n threads denoted by t^i. Threads can execute different functionalities but all they share the same memory address space. They may also use an additional private memory. Each thread t has its own control flow graph CFG^t that is built by using the same concepts of traditional programs [3]. In short, a CFG of a thread t is composed of a set of nodes N^t and a set of edges E_I^t. These edges that link nodes in the same thread are called intra-thread edges. Each node n in the thread t is represented by the notation n_i^t, a well-known terminology in the software testing context. Each node corresponds to a set of commands that are sequentially executed or can be associated to a synchronization primitive (*post* or *wait*).

A multithread program MT is associated with a Parallel Control Flow Graph for Shared Memory ($PCFG_{SM}$), which is composed of both the CFG^t (for $0 \leq t < n$) and the representation of the synchronization among threads. N and E represent the set of nodes and edges of the $PCFG_{SM}$, respectively. For construction of the $PCFG_{SM}$, it is assumed that (1) n is fixed and known at compilation time; (2) there is implicit communication by means of shared variables; (3) there is explicit synchronization using semaphores (which has two basic atomic primitives: *post* (or p) and *wait* (or w)); and (4) initialization and finalization of threads act as a synchronization over a virtual semaphore.

Three subsets of N are defined: N_t (nodes in the thread t), N_p (nodes with *post* primitives) and N_w (nodes with *wait* primitives). For each $n_i^t \in N_p$, a set $M_w(n_i^t)$ is associated, such that for each $n_i^t \in N_p$, with a *post* to a semaphore *sem*, we define $M_w(n_i^t)$ as the set of nodes $n_j^q \in N_w$, such that exist a thread $q \in [0..n-1]$ and a *wait* primitive with respect to (w.r.t.) *sem* in n_j^q. In a similar way, for each $n_i^t \in N_w$, a set $M_p(n_i^t)$ is associated, such that for each $n_i^t \in N_w$, with a *wait* to a semaphore *sem*, we define $M_p(n_i^t)$ as the set of nodes $n_j^q \in N_p$, such that exist a thread $q \in [0..n-1]$ and a *post* primitive w.r.t. *sem* in n_j^q. In other words, $M_w(n_i^t)$ contains all possible *wait* nodes that can match with n_i^t and $M_p(n_i^t)$ contains all possible *post* nodes that can match with n_i^t.

Using the above definitions, we also define the set $E_S \subset E$ that contains edges that represent the synchronization (edge-s) between two threads, such that:

$$E_S = \{(n_j^t, n_k^q) \mid n_j^t \in M_p(n_k^q) \wedge n_k^q \in M_w(n_j^t)\} \quad (1)$$

The concurrent program shown in the Fig. 1 is used to illustrate these definitions. This program implements the producer-consumer problem with limited buffer, using PThreads library in ANSI C. There are three threads: (1) a master,

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t  mutex, empty, full;
int queue[2], avail;

void *producer(void);
void *consumer(void);

/*Thread T0*/
int main(void) {
    pthread_t prod_h, cons_h;

    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,
                   producer, NULL);/*8*/
    pthread_create(&cons_h, 0,
                   consumer, NULL);/*9*/
    pthread_join(prod_h, 0);            /*10*/
    pthread_join(cons_h, 0);            /*11*/
    exit(0);                            /*12*/
}
```

```c
/*Thread T1*/
void *producer(void) {              /*1*/
    int prod = 0, item;             /*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*/
}
/*Thread T2*/
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("consumed: %d\n",
                        my_item);   /*11*/
    }
    pthread_exit(0);                /*12*/
}
```

Fig. 1. Producer-Consumer implemented with PThreads/ANSI C

which initializes variables and creates the producer and consumer threads; (2) a producer, which populates the buffer; (3) a consumer, which removes items from the buffer for further processing. Table 1 contains values of all sets introduced above.

Figure 2 shows the $PCFG_{SM}$ for the program in the Fig. 1. t^0, t^1 and t^2 represent the master, producer and consumer threads, respectively. Dotted lines represent synchronization edges. Some examples of synchronization edges are: $(9^2, 6^1)$ is a synchronization over semaphore, $(9^0, 1^2)$ is a synchronization of initialization and $(12^1, 10^0)$ is a synchronization of finalization. Note that there may exist internal synchronization edges, such as $(10^1, 6^1)$ and $(9^2, 5^2)$ in Fig. 2.

A path $\pi^t = (n_1^t, n_2^t, ..., n_j^t)$, where $(n_i^t, n_{i+1}^t) \in E_I^t$, is intra-thread if it has no synchronization edges. A path that includes at least one synchronization edge is called an inter-thread path and is denoted by $\Pi = (PATHS, SYNC)$, where $PATHS = \{\pi^1, \pi^2, ..., \pi^n\}$ and $SYNC = \{(p_i^t, w_j^q) \mid (p_i^t, w_j^q) \in E_S\}$ [7]. Here p_i^t is a *post* node i in thread t and w_j^q is a *wait* node j in thread q.

$PCFG_{SM}$ also captures information about data flow. Besides local variables, multithread programs have more two special types of variables: (1) shared variables, used for communication; and (2) synchronization variables, used by semaphores. V denotes all variables. $V_L^t \subset V$ contains local variables of thread t. $V_C \subset V$ contains the shared variables and $V_S \subset V$ contains the synchronization variables. Therefore, we define: $def(n_i^t) = \{x \mid x \text{ is a variable defined in } n_i^t\}$.

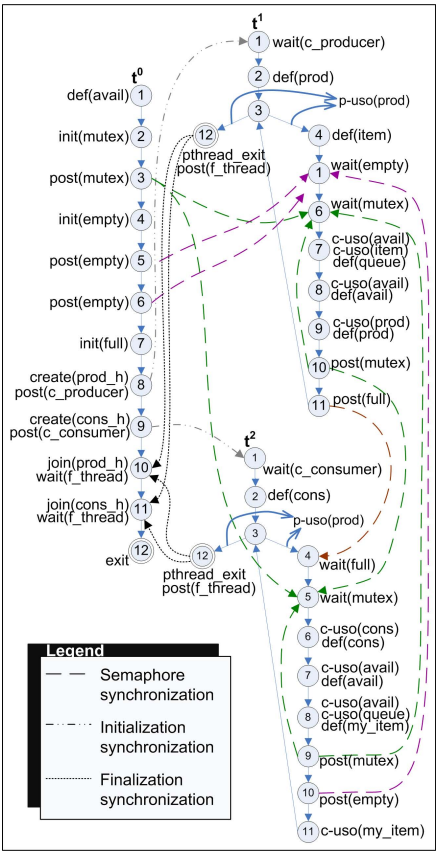

Fig. 2. PCFG_{SM} graph that represents the program shown in Fig. 1

Table 1. Sets of the test model introduced for the program shown in Fig. 1

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

A path $\pi^t = (n_1, n_2, ..., n_j, n_k)$ is definition-clear w.r.t. a local variable $c \in V_L^t$ from n_1 to node n_k or edge (n_j, n_k), if $x \in def(n_1)$ and $x \notin def(n_i)$, for $i \in [2..j]$. The notion definition-clear path is not applicable to shared variables because the communication (definition and use of shared variables) in threads is implicit. It is hard to establish a path that statically defines and uses shared variables. In Secion 2.1, we present a method to determine *execution-based* definition-clear paths for shared variables using a post-mortem methodology.

The use of variables in multithread programs can be: ***computational use (c-use)***: computational statements related to local variable $x \in V_L^t$; ***predicative use (p-use)***: conditional statements that modify the control flow of the thread and are related to local variable $x \in V_L^t$; ***synchronization use (sync-use)***: synchronization statements on semaphores-variable $x \in V_S$;***communicational c-use (comm-c-use)***: computational statements related to shared variable

$x \in V_C$; and ***communicational p-use (comm-p-use)***: conditional statements used on control flow of the thread, related to shared variable $x \in V_C$.

Based on these definitions, we establish associations between variable definition and use. Five kinds of associations are defined: ***c-use association***: is defined by a triple (n_i^t, n_j^t, x) iff $x \in V_L^t$, $x \in def(n_i^t)$, n_j^t has a c-use of x and there is at least one definition-clear path w.r.t. x from n_i^t to n_j^t. ***p-use association***: is defined by a triple $(n_i^t, (n_j^t, n_k^t), x)$ iff $x \in V_L^t$, $x \in def(n_i^t)$, (n_j^t, n_k^t) has a p-use of x and there is at least one definition-clear path w.r.t. x from n_i^t to (n_j^t, n_k^t). ***sync-use association***: is defined by a triple $(n_i^t, (n_j^t, n_k^q), sem)$ iff $sem \in V_S$, (n_j^t, n_k^q) has a sync-use of sem and there is at least one definition-clear path w.r.t. sem from n_i^t to (n_j^t, n_k^q). ***comm-c-use association***: is 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 has a c-use of the shared variable x. ***comm-p-use association***: is 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) has a p-use of the shared variable x.

2.1 Applying Timestamps to Determine Implicit Communication

In this section, we present a method to establish pairs of definition and use of shared variables. These pairs are obtained after execution of the multithread program identifying the order that the concurrent events happened. Lamport [11] presented a way to order concurrent events by means of a happens-before relationship. This relationship can determine if an event e_1 occurs before an event e_2, denoted by $e_1 \prec e_2$.

To obtain this happens-before relationship it is necessary to assign timestamps to concurrent events. Lie and Carver [6] presented a method to assign timestamps that use local logical clock. We adapt this method to assign timestamps in our testing method. The method obtains all synchronizations that happened for an execution and thus generates the communication events.

The method assigns a local logical clock vector, denoted by $t^i.cv$, for each thread t^i. This vector has dimension n, where n is the total number of threads. Each position $i \in [0..n-1]$ on the clock vector is associated to thread t^i. The clock-vector position i is updated when a new event occurs in thread t^i. For instance, observe the c_1 event in t^0 (before the clock vector was $[0,0,0]$). When a synchronization event occurs in t^j other i positions, for $i \neq j$, of the clock-vector can also be updated. For instance, considering the match (p_2, w_2). Before this synchronization the clock vector associated with t_2 was $[0,0,0]$. After, the values were updated to $[2,4,1]$.

The logical space-time diagram shown in the Fig. 3 illustrates the method, using a hypothetical example. This diagram only considers events of synchronization (p_i and w_j) and communication (c_k). Vertical lines represent the logical time of each thread. Arrows among threads represent synchronization events matching *post* and *wait* events. For instance, wait events w_1 and w_2 race the same post event p_1, but the match (w_1, p_1) has occurred.

It is possible that a *wait* primitive has several *posts* to match. These *posts* are inserted in a queue. Our method considers the access criterion *LIFO* to get the happens-before relationship. We chose *LIFO* to get most updated timestamps.

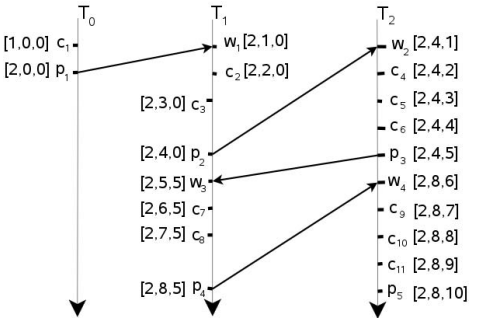

Fig. 3. Example of logical space-time diagram

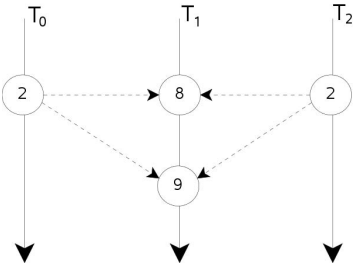

Fig. 4. Example of nondeterminism

Rules defined in [6] are used to establish if an event e_1 happens-before an event e_2. These rules are not showed here for sake of space. With this method, it is possible to show the communications that happened for a program execution.

3 Coverage Criteria

Based on the control, data and communication flow models and definitions presented in previous section, we propose two sets of structural testing criteria for shared-memory parallel programs. These criteria allow the testing of sequential and parallel aspects of the programs.

Control Flow and Synchronization-based Criteria

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

Data Flow and Communication-based Criteria

All-def-comm criterion: the test set must execute paths that cover an association comm-c-use or comm-p-use for all definition of $x \in V_c$.
All-def criterion: the test set must execute paths that cover an association c-use, p-use, comm-c-use or comm-p-use for all definition of $x \in def(n_i^t)$.
All-comm-c-use criterion: the test set must execute paths that cover all comm-c-use associations.
All-comm-p-use criterion: the test set must execute paths that cover all comm-p-use associations.
All-c-use criterion: the test set must execute paths that cover all c-use associations.
All-p-use criterion: the test set must execute paths that cover all p-use associations.

All-sync-use criterion: the test set must execute paths that cover all sync-use associations.

It is necessary to know which path was exercised to evaluate the required elements covered from an execution. One option to obtain this information is to instrument the source code to produce execution trace. This instrumentation can change the original program behaviour. However, this interference does not affect the structural testing proposed here, because it does not prevent the extraction and the future execution of all possible pairs of synchronization.

Due to non-determinism, executions of a program with the same input can cause different event sequences to occur. The Fig. 4 shows the example where the nodes 8^1 and 9^1 in t^1 have non-deterministic *waits* and in nodes 2^0 (t^0) and 2^2 (t^2) have *post* to t^1. All these operations are on the same semaphore. This case illustrates the possible synchronizations among these threads. During the testing activity is essential to guarantee that these synchronizations are executed. Controlled execution is a mechanism used to achieve deterministic execution, i.e. two executions of the program with the same input are guaranteed to execute the same instruction and the specified synchronization sequence. The controlled execution used in this work was adapted from Carver method [12].

The Table 2 shows some required elements for the criteria defined in this section. These required elements are taken on the static analysis.

Table 2. Some required elements by the proposed criteria for the program of the Fig. 1

Criteria	Required Elements	Total
All-nodes-p	$3^0, 5^0, 6^0, 8^0, 9^0, 10^1, 11^1, 12^1, 9^2, 10^2, 12^2$	11
All-nodes-w	$10^0, 11^0, 1^1, 5^1, 6^1, 1^2, 4^2, 5^2$	8
All-nodes	$1^0, 2^0, 3^0, ..., 12^0, 1^1, 2^1, ..., 12^1, 1^2, 2^2, ..., 12^2$	36
All-edges-s	$(3^0, 6^1), (3^0, 5^2), (5^0, 5^1), (6^0, 5^1), (8^0, 1^1), (9^0, 1^2),$ $(10^1, 5^2), (10^1, 6^1), (11^1, 4^2), (12^1, 10^0), (12^1, 11^0), (9^2, 6^1), ...$	16
All-edges	$(1^0, 2^0), (2^0, 3^0), ..., (11^0, 12^0), (1^1, 2^1), (2^1, 3^1), ..., (11^1, 3^1), (3^1, 12^1),$ $(1^2, 2^2), ..., (11^2, 3^2), (3^2, 12^2), (3^0, 6^1), (3^0, 5^2), ..., (9^0, 1^2), (10^1, 6^1), ...$	51
All-def-comm	$(1^0, 7^1, avail), (8^1, 7^1, avail), (7^2, 8^2, avail), (7^1, 8^2, queue)$	4
All-def	$(1^0, 8^2, avail), (2^1, (3^1, 4^1), prod), (4^1, 7^1, item), (7^1, 8^2, queue),$ $(8^1, 7^1, avail), (6^2, (3^2, 12^2), cons), ...$	10
All-comm-c-use	$(1^0, 7^1, avail), (1^0, 7^2, avail), (7^2, 7^2, avail),$ $(7^2, 8^1, avail), (7^1, 8^2, queue), ...$	13
All-comm-p-use	$\emptyset$	0
All-c-use	$(8^1, 7^2, avail), (7^2, 8^2, avail), (2^1, 9^1, prod), (9^1, 9^1, prod),$ $(4^1, 7^1, item), (7^1, 8^2, queue), (8^2, 11^2, my_i tem)...$	16
All-p-use	$(2^1, (3^1, 4^1), prod), (2^1, (3^1, 12^1), prod), (6^2, (3^2, 4^2), cons),$ $(6^2, (3^2, 12^2), cons), ...$	8
All-sync-use	$(2^0, (3^0, 6^1), mutex), (2^0, (3^0, 5^1), mutex), (5^0, (6^0, 5^1), empty),$ $(6^1, (10^1, 6^1), mutex), (5^2, (9^2, 6^1), mutex), ...$	14

4 Case Study

In order to illustrate the proposed testing criteria, consider the program in Fig. 1. The buffer is limited to two produced/consumed items. Due to threads scheduling, two executions are possible: (1) produce, consume, produce, consume

(*PCPC*); (2) produce, produce, consume, consume (*PPCC*). Using controlled execution, it is possible to force the order these executions. Considering to first execution (PCPC) the executed paths and their synchronizations are:

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

For this execution, some coverd elements are the edges-s $(6^0, 5^1)$ and $(12^2, 11^0)$, the comm-c-use $(1^0, 7^1, avail)$, $(7^2, 8^1, avail)$, $(7^1, 8^2, queue)$.

To illustrate how the testing criteria can contribute to reveal faults, consider that the mutex semaphore was initialized with the value 0 or 2 on the main function (code line 17). This will cause a deadlock state or an inappropriate concurrent access to shared variables respectively. An execution that covers the required elements $(3^0, 6^1)$ and $(1^0, 7^2, avail)$, edges-s and comm-c-use respectively, will reveal the fault for the deadlock case. The execution of the required element comm-c-uses $(8^1, 8^2, queue)$ will reveal the fault for the case of inappropriate concurrent access. For both cases other required elements can also reveal these faults.

To illustrate a communication fault consider that *avail* was initialized with 1 (1^0) and all synchronizations are correct. This fault can be revealed with the execution of the required elements comm-c-use $(7^2, 7^2, avail)$ and edge-s $(9^2, 5^2)$. It will be necessary the execution of the PPCC sequence to reveal this fault, since the paths executed with the PCPC sequence do not reveal it.

5 Conclusion

Concurrent programs testing is not a trivial activity. This paper contributes in this context by addressing some of these problems for semaphore-based multithreading programs. The paper introduced both structural testing criteria to validate shared-memory parallel programs and a new model test to capture information about control, data, communication and synchronization from these programs. The paper also presents a post-mortem method based on timestamps to determine which communications (related with shared variables) happened in an execution. This information is important to establish the pairs of definition and use of the shared variables.

The proposed testing criteria are based on models of control and data flow and include the main features of the most used PThreads/ANSI C programs. The model considers communication, concurrency and synchronization faults among threads and also fault related to sequential aspects of each thread.

The use of the proposed criteria contributes to improve the quality of the test cases. The criteria offer a coverage measure that can be used in two testing procedures. Firstly, for generation of test cases, where these criteria can be used as guideline for test data selection. Secondly, for the evaluation of a test set. The

criteria can be used to determine when the testing activity can be ended and also to compare test sets.

The evolution of our work on this subject is directed to several lines of research: 1) development of a supporting tool for the introduced testing criteria (it is now being implemented); 2) development of experiments to refine and evaluate the testing criteria; 3) implementation of mechanisms to validate multithread programs that dynamically create threads; and 4) conduction of an experiment to evaluate the efficacy of the generated test data against *ad hoc* test sets.

References

[1] Yang, C.D., Pollock, L.L.: The challenges in automated testing of multithreaded programs. In: 14th Int. Conference on Testing Computer Software, pp. 157–166 (1997)
[2] Dongarra, J.J., Walker, D.W.: The quest for petascale computing. Computing in Science and Engineering 03(3), 32–39 (2001)
[3] Rapps, S., Weyuker, E.: Selecting software test data using data flow information. IEEE Transactions on Software Engineering SE-11(4), 367–375 (1985)
[4] Taylor, R.N., Levine, D.L., Kelly, C.D.: Structural testing of concurrent programs. IEEE Trans. on Software Engineering 18(3), 206–215 (1992)
[5] Yang, C.S.D., Souter, A.L., Pollock, L.L.: All-du-path coverage for parallel programs. In: Young, M. (ed.) ISSTA 1998: Proc. of the ACM SIGSOFT Int. Symposium on Software Testing and Analysis, pp. 153–162 (1998)
[6] Lei, Y., Carver, R.: Reachability testing of concurrent programs. IEEE Trans. on Software Engineering 32(6), 382–403 (2006)
[7] Vergilio, S.R., Souza, S.R.S., Souza, P.S.L.: Coverage testing criteria for message-passing parallel programs. In: 6th LATW, Salvador, Ba, pp. 161–166 (2005)
[8] Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for testing multi-threaded Java programs. Concurrency and Computation: Practice and Experience 15(3–5), 485–499 (2003)
[9] Lei, Y., Carver, R.H., Kacker, R., Kung, D.: A combinatorial testing strategy for concurrent programs. Softw. Test., Verif. Reliab. 17(4), 207–225 (2007)
[10] Yang, C.S.D., Pollock, L.L.: All-uses testing of shared memory parallel programs. Softw. Test, Verif. Reliab. 13(1), 3–24 (2003)
[11] Lamport, L.: The implementation of reliable distributed multiprocess systems. Computer Networks 2, 95–114 (1978)
[12] Carver, R.H., Tai, K.C.: Replay and testing for concurrent programs. IEEE Softw. 8(2), 66–74 (1991)