The Challenges in Automated Testing of Multithreaded Programs

Cheer-Sun D. Yang and Lori L. Pollock

Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716
(302) 831-1953

Abstract

To take full advantage of parallel architectures, programmers need to be presented with a good set of software tools for developing, debugging, testing, and maintaining their parallel programs. This paper focuses on the challenges that need to be addressed to provide automatic generation of test cases for multithreaded programs, particularly focusing on a program-based, structural testing approach. The major challenges include dealing with nondeterministic execution for reproducible testing, extending the test data adequacy criteria for timing-related testing, extension of the existing structural testing methodologies for multithreaded programs, and the inclusion of timing-related testing. We demonstrate some of the more intricate issues through a more thorough look at providing all-du-path coverage in multithreaded programs.

1 Introduction

Using software tools to automatically generate test cases, generally referred to as *software testing*, has been studied extensively for *sequential* programs as well as *distributed systems* [23, 3, 28]. Recently, some researchers have studied issues regarding the analysis, testing, and debugging of Ada-like *concurrent* programs [29, 31, 25, 20, 27, 14]. There has also been considerable work in analyzing *parallel* programs for the detection of deadlock and race conditions [1, 6, 5], debugging [16, 10, 18, 21, 19], and optimizing analysis and transformation [8, 4, 24, 12, 22, 9]. However, little work has focused on the automatic generation of test cases for the testing of parallel programs, particularly with a program-based approach.

Multiprocessors are being used increasingly as desktop workstations, server platforms, and parallel computing resources. Simultaneously, networks of workstations are being realized as a cost effective vehicle for parallel computing. In attempt to exploit the potential performance gains of the parallelism offered by these systems, programmers are restructuring applications to enable subtasks of the application to be executed simultaneously with appropriate intertask synchronization and communication.

Users can take one of two approaches to parallelizing their application for these machines. When the user writes the parallel code explicitly using parallel language constructs for process creation, communication, and synchronization, the resulting program is called an explicitly parallel program. An implicitly parallel program is created by taking a user's sequential program and running it through a parallelizing compiler or software tool that automatically or semi-automatically identifies opportunities for parallelism and generates a semantically equivalent parallel program.

The model of explicitly parallel programming that we use in the presentation of this work is multithreaded programs for shared memory multiprocessors. A thread is an independent sequence of execution inside a process. Multiple threads can be dynamically created or deleted at run-time. Conditional variables are used to achieve intertask synchronization. Explicitly parallel language constructs include features to specify thread creation and event synchronization, while communication is achieved through shared variables. In this paper, we use *create, post,* and *wait* to model the thread creation, event deblocking, and event blocking, respectively.

For new paradigms of parallel programming to be exploited to obtain performance gains in parallel systems, tools are needed to aid the programmer in developing, debugging, testing, and maintaining their parallel programs. This paper focuses on the challenges of providing automatic program-based testing of multithreaded programs running on multiple processors. After covering some background on program-based testing, we enumerate the major challenges in structural testing of multithreaded programs and describe the state of the art in each of these directions. We then take a closer look at the problem of providing all-du-path coverage for multhithreaded programs in order to demonstrate

Prepared through collaborative participation in the Advanced Telecommunications/Information Distribution Research Program (ATIRP) Consortium sponsored by the U.S. Army Research Laboratory under Cooperative Agreement DAAL01-96-2-0002.

some of the more suttle issues in structural testing of these programs.

2 Background

There are many taxonomic ways to classify software testing methods. If a testing approach requires analysis of the control and/or data constructs of programs, the testing approach is called *structural testing,* or *white box testing*, as opposed to black box testing. If a program testing approach only takes a program as the input, it is considered *program-based*, whereas if a *formal specification* is available for generating test cases, the software testing is called *specification-based.*

As the first step in structural testing of sequential programs, specific sequences of statements, generally referred to as *paths*, are usually produced based on various criteria. Formally, a *path* is an alternating sequence of vertices and edges starting from the *begin* node of a program flow graph to the *end* node. Symbolic execution is then performed to generate a set of test data associated with these paths. As this set of test data is used to exercise the program, these paths will be "passed through". We say that these paths are *covered by this set of test data.* If a path cannot be covered due to the existence of some conflicting conditions for some nodes in the path, the path is called an *infeasible path.*

Program-based testing approaches have been developed for the testing of sequential programs, including executing every *statement* or every *branch* in a program at least once [2]. These methods are called *all-statement coverage* and *all-branch coverage*, respectively. In general, dynamic testing methodologies specify what needs to be tested, known as *criterion.* For example, if a set of test data causes the program being tested to execute all statements within a program at least once, this set of test data is considered *adequate* with respect to the all-statement test criterion. These criteria are also considered criteria for evaluating the *adequacy of test data.*

For example, the test set T is *statement adequate* for a program P, provided that for every statement in P, there exist some test cases in T that cause the statement to be executed. Since the test set T is a finite set, if all test cases in this set are executed at least once, the specific criterion is satisfied by this test set. Therefore, a *test data adequacy criterion* may also be used to determine whether or not a specific testing can be terminated.

In addition to the all-statement and all-branch coverages for testing sequential programs, another major test data adequacy criteria is all-du-path coverage. This method generates one path for each define-use pair; if a variable is defined in one statement and used in another, the path should pass through both statements. A define-use pair, also known as *define-use association(dua)*, is formally represented by a 3-tuple *(V, D, U)*, where the fields D and U indicate the statement numbers of where the variable V is *defined* and *used*, respectively. The idea behind the all-du-path coverage is to generate paths following the pattern of data used within a program.

3 The Major Challenges

Unfortunately, when parallel programs are included in the domain of testing, many algorithms and testing methods fail without major modification mainly due to the intrinsic nature of parallel language constructs and a lack of standard language models for message passing and shared memory parallel programming. Moreover, when the explicit testing of the use of parallel language constructs is taken into account, new issues are raised. This section discusses the issues that need to be addressed in order to provide automatic generation of test cases for multithreaded programs.

3.1 Nondeterministic Execution

In most parallel programming models, deterministic execution is not guaranteed. Therefore, when a parallel program is executed several times with the same input data, the sequence of statements executed may change. However, one may need to control the sequence of execution for re-running a test, in case unexpected results are produced. After the program is fixed, we may need to rerun the test with the same input and the same scenario of execution. Although this issue is essential to testing parallel programs, the motivation is not to be confused with exposing a fault or generating test cases. The motivation for solving this problem is simply to be able to reproduce a specific execution using the same input data. This is called *reproducible testing.* In the community of debugging and testing concurrent programs, many researchers have presented methods to address this issue.

3.2 Test Data Adequacy Criteria

Figure 1 shows the subsumption hierarchy that represents the partial ordering between the major test data adequacy criteria used in testing sequential programs.

In addition to these test data adequacy criteria, the testing of parallel programs demands additional testing criteria that focus on timing-related testing. In particular, testing of parallel programs needs to consider the execution timing of multiple threads on multiple processors as another variable in testing in addition to

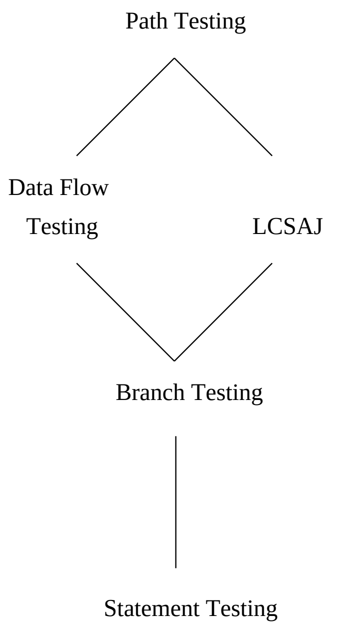


Figure 1: Partial Ordering of Coverage Criteria

test data inputs. For parallel programs, timing-related testing criteria may include synchronization testing, execution sequence testing, and communication testing. As a result, the test data adequacy criteria hierarchy, which is traditionally used for evaluating test data for sequential programs, needs to take these additional criteria into account. The depicted subsumption relation must be extended to include these new testing criteria.

3.3 Use of Existing Testing Methods

Some sequential testing algorithms, e.g., all-du-path and all-branch coverage, fail when they are applied to generate test cases for parallel programs. For instance, in Figure 2, when the variable m is defined in thread T_2, and used in thread T_1, the define-use pair must be covered by at least *two* paths, one in each thread, instead of one path. If we directly apply the all-du-path coverage algorithm for sequential programs, we will obtain two *partial paths* connected by an edge from a *post* to a *wait*. In Figure 2, if we use the original algorithm for generating all-du-path coverage, the path returned is begin-10-loop-11-post-wait-6-endloop-8-end, which really consists of two partial paths. Current symbolic execution approaches cannot use two partial paths to generate test cases.

As a second example, when applying the algorithm for all-branch coverage to parallel programs, the algorithm cannot simply cover all branches in one thread and totally ignore statements in other threads because

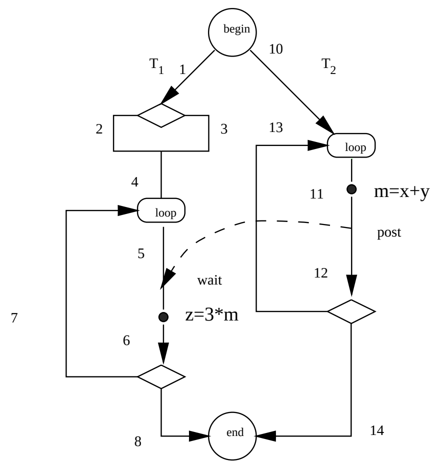


Figure 2: Example of Multiple Threads

some *wait* operations require corresponding *post* statements to be covered. If the sequential algorithm is applied directly, the path produced may cause an infinite wait. For example, in Figure 2, the path begin-1-if-2-endif-4-loop-5-wait-6-endloop-8-end cannot be executed alone without executing another path in a different thread in which the matching *post* is covered. This issue is not easy to resolve since the static analysis must be able to identify the matching nodes of each synchronization call and compute reaching definitions information across threads, while taking into account multiple threads which may be dynamically changing. These examples begin to demonstrate how multiple thread creation, event synchronization, and message passing all complicate test case generation beyond simple extension to current algorithms for generating test cases for sequential programs. Section 5 looks more closely at some of these issues.

3.4 Timing-related Testing

In the paper [11], Gait studied the effect of inserting delay statements into concurrent programs. In particular, the behavior of concurrent programs with synchronization errors was observed. The *probe effect* is seen as an alteration in the frequency of run-time computational errors observed when delays are introduced into concurrent programs. According to Gait's study, the probe effect is observed in the altered behavior of concurrent programs as either a non-functioning concurrent program works with inserted delays, or a functioning concurrent program stops working when previously embedded implicit delays are removed, or relocated, or perhaps changed in value.

Gait's study strongly motivates the need to utilize a testing scheme that changes the relative timing of different events in a parallel program deliberately to uncover potential errors of several kinds: synchronization errors, communication errors, or task creation errors. Based on Gait's experimental results, we can formulate several basic problems in order to apply the delay execution approach effectively for testing parallel programs: (1) the delay duration needs to change, (2) the delay points must cover necessary program locations, and (3) delay statements need to be inserted as well as removed or relocated to induce a probe effect.

One important question needs to be answered: How many delays do we need to insert? The total number of possible test cases per input data set can be formulated as:

$$\binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = 2^n - 1,$$

where n is the total number of synchronization calls, process creation, and communication calls in a parallel program.

This potential exponential number of test cases strongly motivates the need for techniques to reduce the number of timing-related tests. The focus of these techniques needs to be reducing the number of inserted delay statements and the number of different durations of delays tested.

4 Current State of the Art

4.1 Nondeterministic Execution

In [26], Tai, Carver, and Orbid present an approach, known as *deterministic execution* (DET), to deal with non-deterministic execution during debugging and testing of Ada-like concurrent programs. A compiler is provided to insert a conceptual "request for permission" call in front of each rendezvous call. A separate control task runs with the responsibility of controlling the order of events. The transformed program, including the control task, takes both the input data and various synchronization sequences(SYN-sequences) as input to the transformed program and determines the sequence of execution. The control task controls program execution by synchronizing with the original tasks in the program using the SYN-sequences specified by a tester. In general, this approach provides an effective method to force the occurrence of a particular execution sequence for Ada-like concurrent programs which is essential for replay in debugging. However, controlling the order of synchronization event sequences by coordinating with an additional control task when executing multiple threads on multiple processors is a difficult task to achieve.

In [16], LeBlanc and Mellor-Crummey present an approach to record a partial order of the shared object accesses at run-time; at the subsequent re-run, the trace information is used to control the execution sequence. In [21], Netzer and Miller propose a novel strategy to trace and replay message passing parallel programs for debugging purposes. By making run-time tracing decisions, their method traces only a fraction of the total number of messages, gaining two orders of magnitude reduction in trace size over traditional techniques which trace every message, without slowing down the execution speed of replay.

In [29], Weiss proposes the use of serialization to simulate the execution of Ada-like concurrent programs. He motivates the need to find a good representative set of serializations to test parallel programs due to the impractical number of serializations.

Based on all of the previous work in dealing with nondeterminism for reproducible testing, we do not consider this issue further for the automatic generation of test cases for multithreaded programs.

4.2 Test Data Adequacy Criteria

Weiss[29] suggests the need for modification of the notion of "applicability" of existing test data adequacy criteria for the testing of Ada-like concurrent programs. In his paper, Weiss claims, for example, that there are concurrent programs with unreachable statements, therefore *all-statement coverage* is not applicable for testing concurrent programs. Using serialization to simulate a program, he presents a "tree-like" hierarchy of concurrent program-based adequacy criteria derived from sequential program-based criteria. Each different serialization of tasks would correspond to a different possible run of the concurrent program with a fixed set of input data. The main idea is to be able to employ the program-based testing criteria originally used for testing sequential programs to test concurrent programs. However, the implementation of serialization is not viable for testing multithreaded programs running on multiple processors as serialization simulates program execution by a single process.

In [27], Taylor, Levine, and Kelly design a *concurrency graph* to represent the state transitions of a concurrent program. In addition, a set of state-based test data adequacy criteria and a hierarchy of subsumption are presented. A tester may use the concurrency graph as a road map to identify what to test and to control the progress of testing. However, this method suffers a drawback known as the *state explosion prob-*

lem. When there are many synchronization calls in a program, the number of state transitions(edges) in a concurrency graph may grow exponentially. Furthermore, this approach hides the sequential activities in each concurrent state. It may be difficult to find the corresponding path coverage with respect to each state and transition.

In [20], Morasca and Pezzè use a high level *Petri Net*, called ER net, to derive test data adequacy criteria for concurrent and real-time systems. They propose test data adequacy criteria such as *Firing Testing*, *Firing Sequence Testing*, *Transition Testing*, and *Transition Sequence Testing*. Due to the presence of iterations in a program, the transition sequence testing and firing sequence testings may require the execution of an unbounded number of sequences. They propose one intuitive way of extending the class of criteria to require the execution of all possible interleavings of transition firings without repeating the firing of the same transition (1-time testing), or limiting the number of iterations (n-time firing or transition testing), thus tackling the problem of revealing some of the errors which appear only after some iterations, without requiring an unbounded amount of test data. Since some redundant executions, i.e., executions which hardly reveal errors, may be included in the N-time testing, they propose an equivalence class of testing criteria, called *N-Notable Testing*, obtained by clustering equivalent firing sequences. Unfortunately, they only discuss various criteria; the correctness and application are not presented. Similar to using the concurrency graph for generating path coverages, it is not an easy task to relate path coverages to transitions or firing sequences.

4.3 Timing-related Testing

In addition to the deterministic execution method(DET), Tai[25] also suggests *multiple execution* as an approach to dealing with nondeterministic execution. This approach, MET, executes a program multiple times, thus, increasing the possibility of fault finding. Instead of ordering event sequences explicitly, Yang and Pollock[30] have extended and improved Tai's MET method for delaying execution of concurrent programs in order to test the use of parallel language constructs in multithreaded programs. In particular, with the aid of a testing tool, the tester delays various events, i.e., synchronization events, communication events, and thread creations, and executes the program multiple times with various durations for each test case. Thus, various execution schedules are simulated.

As mentioned earlier, there are several issues that need to be addressed: (1) how to achieve delayed execution, (2) where to insert delays, and (3) how many delays are needed for adequate timing-related testing. First, the delayed execution thread will be achieved with the aid of software tools. Hence, the testing process can be semi-automated; the process of inserting *delay statements* or simply *delays* such as sleep(), executing the tests, as well as collecting the results can be done with the aid of a testing tool. However, the results of testing multithreaded programs still must be inspected by human testers.

Second, to answer the question of where delay statements should be inserted, we need to focus on the goal of timing-related tests. The goal of timing-related testing is to detect unpredictable results produced by incorrect coding of synchronization calls, thread creation calls, and/or communication calls. Thus, the potential locations where delay statements can be inserted for the most benefit in testing are locations in front of synchronization, communication, or thread creation calls, or before the first executable statement of a thread. These locations are referred to as *delay points*. In the next subsection, we describe our approach to reducing the number of timing-related test cases potentially generated using this approach.

4.4 Reducing the Cost of Testing

Yang and Pollock[30] have developed a technique for reducing the number of timing-related test cases for multithreaded programs, based on identifying *redundant delays*. A redundant delay is defined informally as a delay point whose effect of delaying execution can be achieved by another delay point.

Redundant delays can be computed by performing a straightforward data flow analysis over the parallel program flow graph. In particular, the set *Reach(n)* is first computed for every delay point, where *Reach(n)* contains all of the delay points that can directly or indirectly reach this delay point along control and/or synchronization edges. By formulating the computation in terms of a data flow analysis framework, the sets can be computed by executing the iterative data flow analysis algorithm.

After the set of reaching delays is computed, this information is used to identify delay redundancy. The set of intrusive delays to a node n, i.e., $Intrusion(n)$, associated with a synch edge can be determined by using the following formulas where the *post* and the *wait* are associated with the same synch edge:

$$Reach_{both} = Reach_{in}(post) \cap Reach_{in}(wait); \quad (1)$$

$$Intrusion(post) = Reach_{in}(post) \; - \; Reach_{both}; \quad (2)$$

$$Intrusion(wait) = Reach_{in}(wait) \; - \; Reach_{both}; \quad (3)$$

In the above equations, the intersection of the set $Reach_{in}(post)$ and $Reach_{in}(wait)$ represents the reaching delays that reach both threads with which the synch edge $(post, wait)$ is associated. In other words, delays in this set can kill neither the delay immediately in front of the *post* nor the delay immediately in front of the *wait*. Since the intrusive source nodes with respect to a *post* and a *wait* are those that are only in $Reach(post)$ or only in $Reach(wait)$, respectively, the intrusive reaching delays for a *post* and a *wait* can be computed by extending Equations 2 and 3, respectively.

That is, d_{post} can be removed because it is redundant if $Intrusion(post) - d_{post}$ is nonempty, where d_{post} is the delay immediately before the *post*. Similarly, d_{wait} can be removed because it is redundant if $Intrusion(wait) - d_{wait}$ is nonempty. Thus, when there exists at least one intrusive source node d_i in the set $Intrusion(n)$ at the node n (i.e., a *post* or a *wait*) and d_i is not the delay point d_n, which is immediately in front of the node n, d_n is redundant.

The delay execution approach has several merits. (1) It is easy to implement on multiple processors. Unlike serialization and deterministic execution which require ordering synchronization events, this approach tests a program with the consideration of timing changes on multiple machines without the need to explicitly order synchronization events. (2) There is no need to generate a concurrency graph. (3) It is effective in terms of fault detection. The fault detection capability of the delay execution approach is virtually the same as that of the deterministic execution approach through multiple execution with random delay execution. (4) The total number of test cases can be reduced by static analysis.

On a similar notion of reducing cost, a perturbation analysis [13] can be performed in attempt to reduce the intrusiveness of program instrumentation for the monitoring and debugging of distributed systems. The goal is to identify the situation in which the run-time monitoring activities can be performed non-intrusively, that is, without affecting the outcome of non-deterministic events. Although the concept of intrusion is exploited in the delay execution work when delay statements are introduced for the testing of parallel language constructs, the major goal of the paper [13] and the delay execution work is different. In the delay execution work, statements of intrusive nature are purposely inserted in front of some parallel language constructs to cause intrusion, whereas the goal of Gupta and Spezialetti's work is to reduce the intrusiveness effects of probes inserted for program monitoring.

4.5 Static Analysis of Parallel Programs

Static program analysis has played a key role in various methodologies for automatic test case generation for sequential programs, and will also play a similar role for multithreaded program testing. There has been a handful of researchers investigating static analysis of parallel programs, all with different objectives, unrelated to automatic generation of test cases.

Srinivasan and Wolfe[24] present a solution to the reaching definitions data flow problem for programs with explicit lexically specified parallel constructs, such as *cobegin/coend* or *parallel sections*, both with and without synchronization operations, such as *post, wait*, or *advance*. In the paper [12], Srinivasan and Wolfe describe semantics for parallel constructs, and introduce new intermediate forms, called the *Parallel Control Flow Graph*, and the *Parallel Precedence Graph*.

Chow and Harrison[4] present a data flow framework for analyzing parallel programs with *cobegin/coend* parallelism without explicit synchronization. Long and Clarke[17] develop a data flow framework for Ada programs with task parallelism and "rendezvous" synchronization, but they do not handle shared variables. Dwyer and Clarke[7] present an approach based on data flow analysis for verifying properties of Ada-like concurrent programs. A specific type of regular expression, e.g., *qualified regular expressions*(QRE), is used for specifying a property of a concurrent program. As an example, one can specify that *after an event a occurs, eventually an event c will occur*. Their data flow based analysis tool will take the concurrent program as well as the QRE's specified by the user and generate analytical results. They claim that their approach has the potential to analyze concurrent programs cost effectively with respect to explicitly stated correctness properties.

In summary, some of these static analyses can be exploited for automatically generating test cases. In particular, solutions to the reaching definitions problem for parallel programs provides necessary information about parallel programs to perform testing analysis for *all-du-path coverage*. They also present solutions to the problem of finding matching synchronization calls.

5 All-du-path Testing

In this section, we give an overview of our work in investigating the applicability of all-du-path testing to multithreaded programs. As one of the major testing criteria, all-du-path testing seeks to cover each define-use association in a program by at least one test case in the test suite. After we describe the program representation used for program analysis and present some necessary terminology, we discuss several issues that need to be addressed in the development of techniques to cover all-du-paths in testing multithreaded programs. We conclude with a description of a set of conditions to be used in judging the effectiveness of all-du-path testing algorithms.

To represent the control flow of a multithreaded parallel program, a *Parallel Program Flow Graph* (PPFG) is defined to be a graph $G = (N, E)$ in which N is the set of nodes representing statements in the program, and E consists of three sets of edges E_S, E_T, and E_I. The set E_I consists of intra-thread control flow edges (m^i, n^i), where m and n are nodes in thread T_i. The set E_S consists of synchronization edges ($post^i$, $wait^j$), where $post^i$ is a *post* statement in thread T_i, $wait^j$ is a *wait* statement in thread T_j, and $i \neq j$. The set E_T consists of thread creation edges (n^i, n^j), where n^i is a call statement in thread T_i to the *create()* function, and n^j is the first statement in thread T_j ($T_i \neq T_j$).

We define a *path* P_i within a thread T_i to be an alternating sequence of nodes and intra-thread edges $n^i_{u_1}$, $e^i_{u_1}$, $n^i_{u_2}$, $e^i_{u_2}$, ..., $n^i_{u_k}$ or simply a sequence of nodes $n^i_{u_1}, n^i_{u_2}, \ldots, n^i_{u_k}$, where u_w is the unique node index in a unique numbering of the nodes and edges in the control flow graph of the thread T_i (e.g., a reverse postorder numbering).

A *du-pair* is a triplet (var, n^i_u, n^j_v), where n^i_u is the u^{th} node in thread T_i in the unique numbering of the nodes in thread T_i, and the program variable *var* is defined in the statement represented by node n^i_u, while the program variable *var* is referenced in the v^{th} node in the unique ordering of nodes in thread T_j.

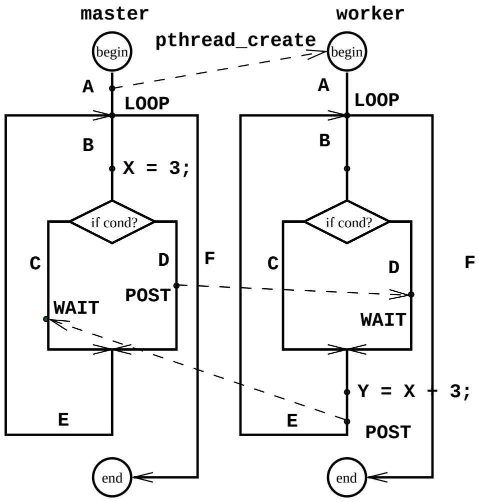
In a sequential program or a single thread T_i, we consider a node n to be *covered by a path* P_i, represented by $n \in_p P_i$, if there exists a node n^i_s in the path P_i such that $n = n^i_s$. We say that a node n^l $(1 \leq l \leq k)$ in a multhithreaded program is *covered by a set of paths* $PATH = (P_1, \ldots P_k)$ in threads T_1, $T_2, \ldots T_k$, respectively, or simply $n^l \in_p PATH$, if $n^l \in_p P_l$.

We represent the set of matching posts of a wait node as $MP(w) = \{p | (p, w) \in E_S\}$ and the set of matching waits of a post node as $MW(p) = \{w | (p, w) \in E_S\}$. We use the symbol "$\prec$" to represent the Lamport's *happen before* relation between two nodes[15]. We say $a \prec b$ if the node a is executed before the node b.

5.1 Issues

In this section, we use some simplified examples to demonstrate some issues to be addressed in finding all-du-paths in multithreaded programs. This list is not necessarily exhaustive, but instead meant to illustrate the complexity of the problem of automatically generating all-du-paths for multithreaded programs.

Figure 3 contains two threads, the `master` thread and a `worker` thread. The dots labeled with letters indicate specific points in the program used to indicate paths. The du-pair to be covered is the definition of x in $x = 3$ of the `master` thread and the use of x in $y = x + 3$ in the `worker` thread. The figure shows one path coverage for this du-pair.



PATH COVERAGE:

MASTER: begin-A-loop-B-if-D-endif-E-loop-F-end

WORKER: begin-A-loop-B-if-D-endif-E-loop-F-end

Figure 3: Du-pair covered, but not matching *wait*.

This figure demonstrates a path coverage that indeed covers the du-pair, but does not cover both the *post* and the *wait* of a matching *post* and *wait*. If the *post* is covered and not the matching *wait*, the program will execute to completion, despite the fact that the synchronization is not covered completely. However, if the *wait* is covered and not the matching *post*, then the program will hang with the particular test case. In the case of Figure 3, the test will run to completion. In figure 4, the path coverage for the `worker` does not include the *post* call when there is a correct path that would issue the *post* call. The paths generated here may cause the `master` task to wait indefinitely.

In figure 5, the generated paths cover both the *define* and the *use* nodes, but the *use* node will be reached before the *define* node, that is, $use \prec define$. If the data flow information reveals that the definition of x in the `worker` thread should indeed be able to reach the use of x in the `master` thread, then we should attempt to find a path coverage that will test this pair. The current path does not accomplish this.

In figure 6, the generated paths cover the du-pair correctly because the *define* node will be executed prior to the *use* node. However, the `worker` task may not

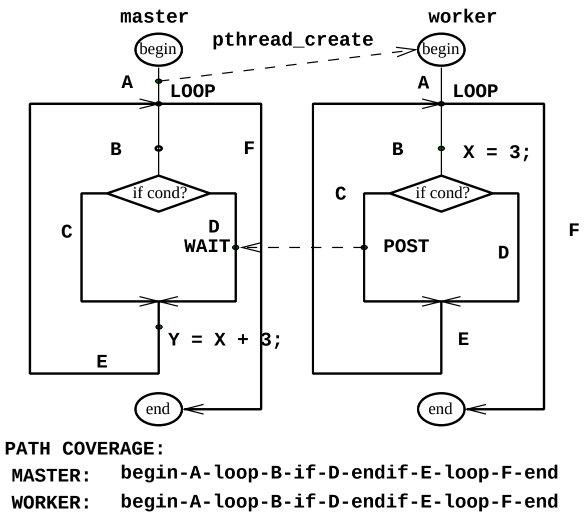


Figure 4: Du-pair covered, but not matching *post*.

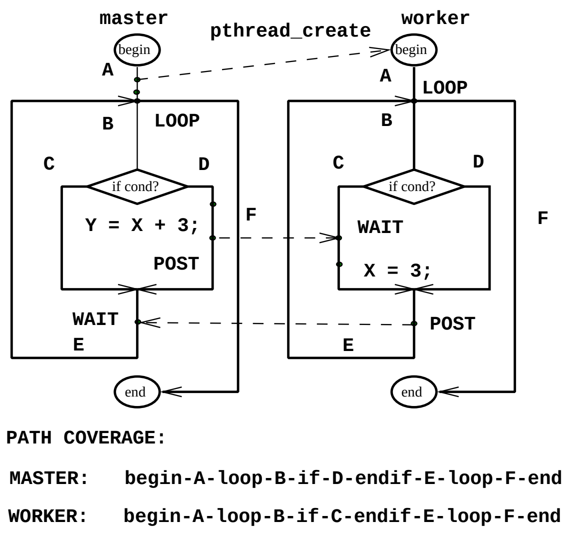


Figure 5: Du-pair is incorrectly covered.

complete execution, whereas the `master` task will terminate successfully. The generated path will cause the loop in the `master` task to iterate only once, while the loop in the `worker` task will iterate twice. This shows how the inconsistency in the number of loop iterations may cause one task to wait infinitely. In summary, finding such du-pair coverages for situations like this where there are loops and matching *post*'s and *wait*'s inside the loops is not straightforward due to the problems seen in covering the matching *post* and *wait* involved in the paths.

In figure 7, we show how branch selection at an *if* node can influence whether or not all tasks will terminate successfully. At the two *if* nodes in this program, there are several ways to choose the branch for the path coverage. However, the two related branches must be chosen or ignored together. Otherwise, the paths may generate undesirable results, namely an nonterminating program.

5.2 Requirements

The examples motivate a classification of all-du-path coverage algorithms. In particular, we classify each du-path coverage generated by an algorithm for producing all-du-path coverage of a multithreaded program as *acceptable* or *unacceptable*, and *t-runnable* or *non-t-runnable*.

5.2.1 Acceptability of a du-path coverage

We call a set of paths $PATH$ an *acceptable du-path coverage*, denoted as $PATH_a$, for the du-pair $(define, use)$ in a multithreaded program, if all of the following conditions are satisfied:

1. $define \in_p PATH$,
2. $use \in_p PATH$,
3. if $\exists (post, wait) \in E_S$, such that $define \prec post \prec wait \prec use$, then $post, wait \in_p PATH$.
4. $\forall n^j \in_p PATH$ where $(n^i, n^j) \in E_T$, $\exists n^i \in_p PATH$.

These conditions ensure that the definition and use are included in the path, and that any *(post,wait)* edge between the threads containing the definition and use, and involved in the data flow from the definition to the use are included in the path. Moreover, for each sink of a thread creation edge, the associated source of the thread creation edge is also included in the path. If any of these conditions is violated, then the path coverage is considered to be *unacceptable*. For instance, in figure 4, the path coverage fails to be an acceptable path coverage because the third condition is not satisfied. Figure 5, where the define and use are covered in reverse order, shows another instance that only satisfies the first two conditions, but fails to satisfy the third condition.

5.2.2 T-runnability of a du-path coverage

We have seen through the examples that a multithreaded program may not terminate under a given path coverage, even when the du-path coverage is acceptable. If a path coverage can be used to generate a test case

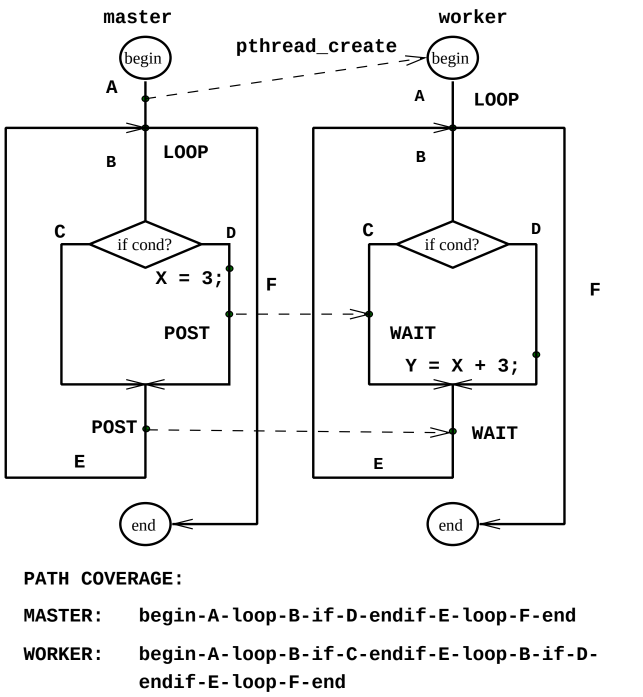


Figure 6: Du-pair coverage may cause an infinite wait.

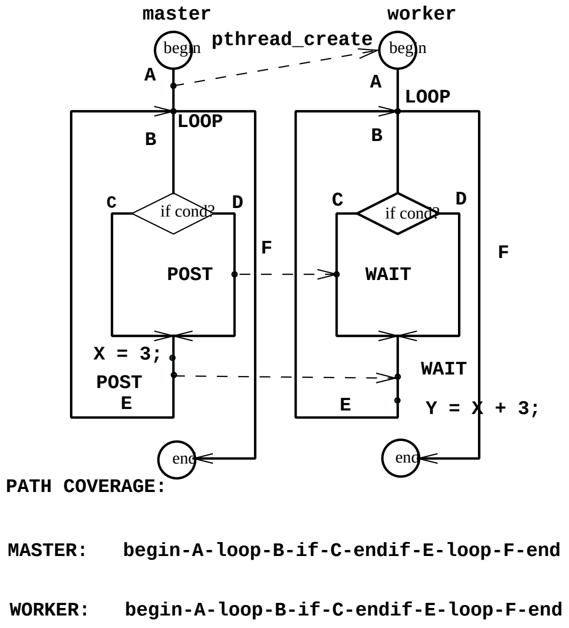


Figure 7: Branch selection sequence can be crucial.

for which the multithreaded program will terminate successfully without causing any infinite wait in any thread, we call the path coverage a *t-runnable du-path coverage*. When a $PATH$ is t-runnable, we represent it as $PATH_t$. More formally, a $PATH_a$ is t-runnable if all of the following additional conditions are satisfied:

1. $\forall$*wait* nodes $w \in_p PATH$, $\exists$ a *post* node $p \in MP(w)$, such that $p \in_p PATH$,
2. for each instance of a *wait*, $w_i^t \in_p PATH$, (possibly represented by the same node $n^t \in_p PATH$), $\exists$ an instance of a *post*, $p_u^s \in_p PATH$, where $p_u^s \in MP(w_i^t)$. An instance of a *wait* or *post* is one execution of the *wait* or *post*; there may be multiple instances of the same *wait* or *post* in the program.

These conditions ensure that, for each instance of a *wait* in the $PATH$, there is a matching instance of a *post*. However, these conditions do not require that for every instance of *post*, a matching *wait* is covered. In other words, the following condition is not required: $\forall$*post* nodes $p \in_p PATH$, $\exists$ a *wait* node $w \in MW(p)$, such that $w \in_p PATH$,

Given a du-pair in a multithreaded program, we claim the following are true without showing formal proofs in this paper:

1. $\exists PATH$ such that $PATH$ is $PATH_a$.
2. Finding a *t-runnable* path coverage $PATH_t$ for any given PPFG, is not computable. In other words, for some multithreaded programs, $PATH_t$ may not exist.
3. Given a path coverage $PATH$ for a du-pair, we can determine whether $PATH$ is $PATH_a$ and/or $PATH_t$.

With these limitations, we can develop algorithms to find $PATH_a$ automatically. However, we utilize user interaction in determining $PATH_t$ in the more difficult cases, and sometimes have to indicate to the user that we cannot guarantee that the execution will terminate on a given test case (i.e., path coverage). In this case, the program can still be run, but may not terminate.

6 Summary and the Future

In this paper, we have described the issues to be addressed in order to provide automatic test case generation for multithreaded programs, similar to the test case generation provided today for sequential programs. As of today, very little effort has concentrated on these issues, but the need for testing methodology and tools for helping the parallel programmer is quickly growing as parallel architectures become more cost effective and more user-friendly parallel programming models are being accepted. The need for automatization will be real, as programs that are parallelized are typically very large, complicated programs, and the inherent nature of parallelism brings new dimensions to the testing and debugging tasks.

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expr essed or implied, of the Army Research Laboratory or the U.S. Government."

References

[1] V. Balasundaram and K. Kennedy. Compile-time detection of race conditions in a parallel program. In *1989 International Conference on Supercomputing*, 1989.

[2] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold Company, 1983.

[3] S. C. Cheung and J. Kramer. Tractable dataflow analysis for distributed systems. *IEEE Transactions on Software Engineering*, 20(8):579–593, Aug. 1994.

[4] J.-H. Chow and W. Harrison. Compile time analysis of parallel programs that share memory. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, 1992.

[5] S. K. Damodaran-Kamal and J. M. Francioni. Testing races in parallel programs with an otot strategy. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 216–227, Aug. 1994.

[6] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms on access anomaly detection. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990.

[7] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. Technical Report CS-1994-045, University of Massachusetts, Department of Computer Science, June 1994.

[8] P. A. Emrath, S. Ghosh, and D. Pandua. Detecting nondeterminacy in parallel programs. *IEEE Software*, pages 69–77, Jan. 1992.

[9] J. Ferrante, D. Grunwald, and H. Srinivasan. Computing communication sets for control parallel progrms. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Pandua, editors, *Languages and Compilers for Parallel Computing, 7th International Workshop Proceedings*, pages 316–330. Springer-Verlag, 1994.

[10] C. J. Fidge. Partial orders for parallel debugging. In *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 183–194, May 1988.

[11] J. Gait. A probe effect in concurrent programs. *Software-Practice and Experience*, 16(3):225–233, Mar. 1986.

[12] D. Grunwald and H. Srinivasan. Efficient computation of precedence information in parallel programs. In *Languages and Compilers for Parallel Computing*, pages 502–616. Springer-Verlag, 1993.

[13] R. Gupta and M. Spezialetti. Towards a non-intrusive approach for monitoring distributed computations through perturbation analysis. In *Languages and Compilers for Parallel Computing*, pages 586–501. Springer-Verlag, 1993.

[14] P. V. Koppol and K.-C. Tai. An incremental approach to structural testing of concurrent software. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 14–23, San Diego, California, June 1996.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–564, July 1978.

[16] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4), Apr. 1987.

[17] D. Long and L. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. Technical Report COINS 91-31, University of Massachusetts, Department of Computer Science, July 1991.

[18] D. C. Marinescu, J. E. L. Jr., T. L. Casavant, and H. J. Siege. A model for monitoring and debugging parallel and distributed software. In *Proceedings of the 13th Annual International Computer Software and Applications Conference*, pages 81–88, Sept. 1989.

[19] J. May and J. D. Choi. A portable, extensible parallel debugger. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 96–106, May 1993.

[20] S. Morasca and M. Pezze. Using high-level petri nets for testing concurrent and real-time systems. In H. Zedan, editor, *Real-Time Systems: Theory and Applications, Proceedings of the conference organized by the British Computer Society*, pages 119–131. Elsevier Science Publishings, 1990.

[21] R. H. Netzer and B. P. Miller. Optimal tracing and replay for debugging a message-passing parallel program. In *Supercomputing*, pages 502–511, Nov. 1992.

[22] V. Sarkar and B. Simons. Parallel program graphs and their classification. In *Languages and Compilers for Parallel Computing*, pages 633–655. Springer-Verlag, 1993.

[23] M. Spezialetti and P. Kearns. Simultaneous regions: A framework for the consistent monitoring of distributed computations. In *Proceedings of the Ninth IEEE International Conference on Distributed Computing Systems*, pages 61–68, 1989.

[24] H. Srinivasan and M. Wolfe. Analyzing programs with explicit parallelism. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Pandua, editors, *Languages and Compilers for Parallel Computing*, pages 405–419. Springer-Verlag, 1992.

[25] K. C. Tai. Testing of concurrent software. In *Proceedings of the 13th Annual International Computer Software and Applications Conference*, pages 62–64, Sept. 1989.

[26] K. C. Tai, R. H. Carver, and E. E. Obaid. Debugging concurrent Ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, SE-17(1):45–63, Jan. 1991.

[27] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, Mar. 1992.

[28] S. Venkatesan and B. Dathan. Testing and debugging distributed programs using global predicates. *IEEE Transactions on Software Engineering*, 21(2):163–177, Feb. 1995.

[29] S. N. Weiss. A formal framework for studying concurrent program testing. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 106–113, July 1988.

[30] C.-S. Yang and L. L. Pollock. Identifying redundant test cases for testing explicitly parallel language constructs. *ARL-ATIRP First Annual Technical Conference*, January 1997.

[31] M. Young, R. N. Tylor, K. Forester, and D. Brodbeck. Integrated concurrent analysis in a software development environment. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 200–209, Dec. 1989.