# All-uses Testing of Shared Memory Parallel Programs

*Cheer-Sun D. Yang*
Computer Science Department
West Chester University of PA
West Chester, PA 19383
(610) 738-0450
yang@coyote.cs.wcupa.edu

*Lori L. Pollock*
Computer and Information Sciences
University of Delaware
Newark, DE 19716
(302) 831-1953
pollock@cis.udel.edu

## Abstract

Parallelism has become the way of life for many scientific programmers. A significant challenge in bringing the power of parallel machines to these programmers is providing them with a suite of software tools similar to the tools that sequential programmers currently utilize. Unfortunately, writing correct parallel programs remains a challenging task. In particular, automatic or semi-automatic testing tools for parallel programs are lacking. This paper takes a first step in developing an approach to providing all-uses coverage for parallel programs. A testing framework and theoretical foundations for structural testing are presented, including test data adequacy criteria and hierarchy, formulation and illlustration of all-uses testing problems, classification of all-uses test cases for parallel programs, and both theoretical and empirical results with regard to what can be achieved with all-uses coverage for parallel programs.

## 1 Introduction

As computational scientists continue to demand higher performance, the use of parallelism is becoming more pervasive. A variety of parallel programming languages and sophisticated compiler technology has been developed for gaining efficient use of the available parallelism.

Unfortunately, writing correct parallel programs remains a challenging task. First, within a single parallel program, multiple tasks may be executed in parallel. It is more difficult for a programmer to keep track of the exact execution paths of a program with multiple tasks. Hence, verifying the results becomes more difficult. Second, data communication and synchronization among threads are frequently needed. A programmer must consider not only the logic within one thread, but also the data communication and synchronization among threads. Finally, when synchronization errors or race conditions exist, nondeterministic execution can cause the results of one execution run to be different from another run even using the same input data. Due to nondeterministic execution, a "bug" in a parallel program may not be detected even when the program is executed multiple times. A major obstacle to users in ensuring the correctness and reliability of parallel software is the current lack of software testing tools for the parallel programming paradigm.

Researchers have proposed methodologies and algorithms for generating test cases automatically for testing sequential programs [1, 2, 3] as well as distributed systems [4, 5]. Also some researchers have studied the problem in the context of concurrent program testing [6, 7, 8, 9]. Other work has focused on the detection of race conditions or debugging [10, 11, 12]. The goal of many of these efforts has been to be able to reproduce a test run. Many tools which incorporate these methodologies have been developed. Automatic and semi-automatic testing tools have aided sequential and concurrent software developers during and after the stage of software development. Various software testing methodologies and tools have provided programmers with systematic ways to perform system testing. However, because the intrinsic nature of parallel programs significantly complicates the task of providing program-based testing tools to developers of parallel programs, few tools are actually available.

The main objective of this effort is to develop software testing tools to semi-automate the structural testing of parallel programs. This paper takes the first step in developing such tools for all-uses testing coverage of parallel programs. A testing framework and theoretical foundations for structural testing are presented, including test data adequacy criteria and hierarchy, formulation and illlustration of all-uses testing problems, classification of all-uses test cases for parallel programs, and both theoretical and empirical results with regard to what can be achieved with all-uses coverage for parallel programs.

The remainder of this paper is organized as follows. Section 2 defines the program representation called Parallel Program Flow Graph(PPFG), for modeling the parallel program for static analysis. Section 3 defines a testing framework including the the all-uses testing paradigm, structural testing data adequacy criteria and a testing procedure for dealing with the nondeterministic nature of shared memory programs. Section 4 describes inherent problems of all-uses testing of parallel programs, and defines a classification of all-uses test cases. Section 5 presents both theoretical and empirical results with regard to what can be achieved with all-uses coverage for parallel programs. Related work is discussed in Section 6. Section 7 describes conclusions and outlines future work.

## 2 Parallel Program Representation and Terminology

This work focuses on programs that have been written explicitly as parallel programs. While sophisticated optimization and parallelizing compiler technology can exploit parallelism to improve the performance of sequential programs, these tools are limited by the underlying sequential algorithm. The targeted programs are those with thread creation and event synchronization to support the MIMD programming style; however, extensions to the representation for SPMD style programming are given in [13]. In this work, a shared memory parallel program is defined as follows.

**Definition: A Shared Memory Parallel Program** $PROG$

A shared memory parallel program $PROG$ written in language $\mathcal{L}$ is defined as a set of threads of execution, or simply $\mathcal{P}ROG = \{T_1, T_2, \ldots, T_n\}$, where $T_i, (1 \leq i \leq n)$ represents the $i^{th}$ thread, and there are $n(\geq 2)$ threads. $T_1$ is called the *manager* thread created upon the start of execution of the program. All other threads are called *worker* threads, which are created by a function call for thread creation from the manager thread. Communication between threads is achieved through shared variables. Condition variables are used to enable one thread to wait on another thread; *post* and *wait* operations are the two basic operations on a condition variable. When a thread `t` executes a *wait* operation on a given condition variable, `t`'s execution is blocked, (i.e., the thread `t` is put into a sleep state). When a thread `s` executes a *post* operation on a given condition variable, either one or all of the threads with a pending *wait* on that condition variable are awakened, depending on the arguments of the *post* operation.

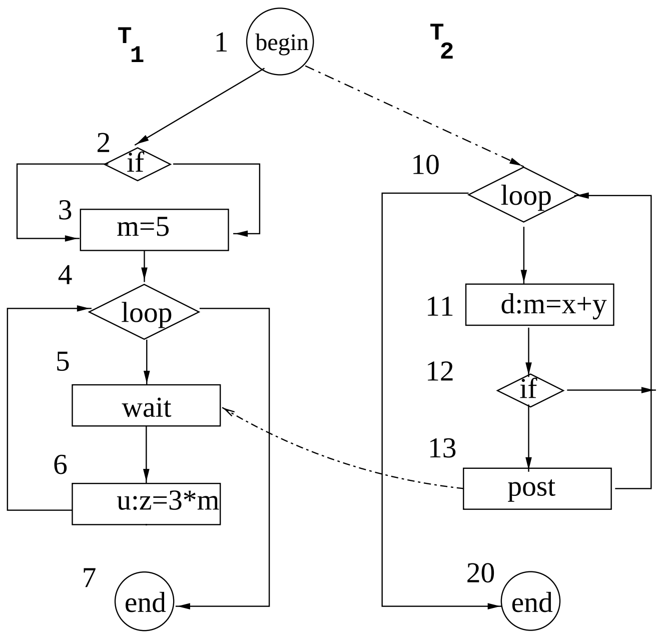


Figure 1: Simple Example of PPFG

Static analysis of a program is typically performed on a graphical representation of the program as it is much easier to perform the analysis on a graph that depicts the control flow of the program explicitly. To represent the control flow of a parallel program written in $\mathcal{L}$, a graphical representation, called a *Parallel Program Flow Graph*($PPFG$), is defined as follows.

**Definition: Parallel Program Flow Graph(**$PPFG$**)**

A PPFG is a graph $G = (V, E)$, where $V$ is the set of nodes, each representing a statement in the source program. The set $V$ can be partitioned into $n$ sets of nodes $V_1, V_2, \ldots, V_n$, where all nodes in $V_i$ are located in thread $T_i$, $1 \leq i \leq n$. $E$ is the set of edges, partitioned into $E_S$, $E_T$, 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 ($p^i_s$, $w^j_t$), where $p^i_s$ is a *post* statement with $s$ as the node id in thread $T_i$, $w^j_t$ is a *wait* statement with $t$ as the node id in thread $T_j$. The set $E_T$ consists of thread creation edges ($n^i$, $n^j$), where

$n^i$ is a call in thread $T_i$ to the thread creation function, and $n^j$ is the first statement in thread $T_j$ $(i \neq j)$.

As an example, Figure 1 illustrates a simple example of a PPFG. All solid edges are intra-thread edges in $E_I$. Edges in $E_S$ and $E_T$ are represented by dotted edges. In this example, $m$ is a shared variable. In Figure 1, a circle represents a *begin* or an *end* node, a diamond represents an *if*- or a *loop*-node, and a rectangle represents other program statements. Within each thread, each statement is numbered with a reverse postorder number. This graph representation is different from traditional control flow graphs for sequential programs in two ways: (1)*if* and *loop* nodes are distinguished from other nodes, and (2) thread creation and synchronization edges are distinguished from other control transitions for several reasons: the static analysis of a shared memory parallel program requires uniquely identifying communication nodes in order to apply different analysis, known as *transition functions*, and the path identifying algorithm requires different data structures for *if* and *loop* nodes. Under some situations, the reverse postorder number is used to identify the branch to take at a *loop* node. When a shared memory parallel program is represented as a PPFG, some assumptions are implicitly made in order to be able to statically model the program and enable static analysis on the model:

- The total number of tasks to be created is known at compile-time.
- The programming paradigm is assumed to be in MIMD style, and not in SPMD style.
- The potential matching *post* and *wait* nodes are assumed to be identifiable statically using the event id associated with the *post* and *wait* calls. Hence, the event id cannot be dynamically assigned.

These assumptions can be relaxed by duplicating the representation of parallel code segments and more extensive program analysis; the extensions are described in detail in [13]. For instance, the MIMD model is able to simulate the SIMD and SPMD models. Similarly, the total number of tasks may not have to be known if a representative structure can be built for the different sections of code that would be run by different threads. Static data flow analyses could be performed to reduce the set of possible values of event ids for given *post* and *wait* calls, in order to build a conservative, but reasonable size graph. A *post* that could take on any dynamically assigned value would result in possibly matching any *wait* in the program, if no static analysis is performed to reduce the possible values and matching *wait*'s.

**Definition: A Path in a PPFG**

Given a PPFG, a *path* $P_i(n^i_{u_1}, n^i_{u_k})$ or simply $P_i$, within thread $T_i$ is defined 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 in the control flow graph of the thread $T_i$ (e.g., a reverse postorder numbering).

In this paper, the presence of a node in a path is called an *instance* of the node. For example, in Figure 1, the path 1-2-3-4-5-6-4-7 covers two instances of node 4.

**Definition: A Complete/Partial Path in a PPFG**

Given a PPFG and a *path* $P_i(n^i_{u_1}, n^i_{u_k})$, $P_i$ is called a *complete path* if $n^i_{u_1}$ is a *begin* node and $n^i_{u_k}$ is an *end* node. If either $n^i_{u_1}$ is not a *begin* node but $n^i_{u_k}$ is an *end* node, or $n^i_{u_1}$ is a *begin* node but $n^i_{u_k}$ is not an *end* node, $P_i$ is called a *partial path.*

In Figure 2, the manager thread creates two worker threads. An example of a complete path in the *manager* thread is 1-2-3-4-5-6-8-9-3-11. An example of a complete path in $worker_1$ is 21-22-23-25-26-27-28-22-23-24-27-28-22-30. In the remainder of this paper, unless otherwise stated, the term *path* is assumed to mean a complete path.

**Definition: A Connected Sequence**

A *connected sequence*, represented as $CS(n^i_{u_1}, n^j_{u_k})$, is defined to be an alternating sequence of nodes and edges, which could be intra-thread or synchronization edges from $n^i_{u_1}$ in thread $T_i$ to $n^j_{u_k}$ in thread $T_j$, where $u_w$ is the reverse post order number in the control flow graph of each thread $T_i$ and $T_j$. A connected sequence that involves only one synchronization edge is: $n^i_{u_1}$, $e^i_{u_1}$, $n^i_{u_2}$, $e^i_{u_2}$, …, $n^i_{u_s}$, $e^{i,j}_{u_s,u_t}$, $n^j_{u_t}$, $e^j_{u_t}$, …, $n^j_{u_k}$ or simply the sequence of nodes $n^i_{u_1}, n^i_{u_2}$, …, $n^i_{u_s}$, $n^j_{u_t}$, …, $n^j_{u_k}$, where $e^{i,j}_{u_s,u_t} = (n^i_{u_s}, n^j_{u_t}) \in E_S$.

In Figure 1, the path 1-2-3-4-5-6-4-7 can be represented as CS(begin, end), while the connected

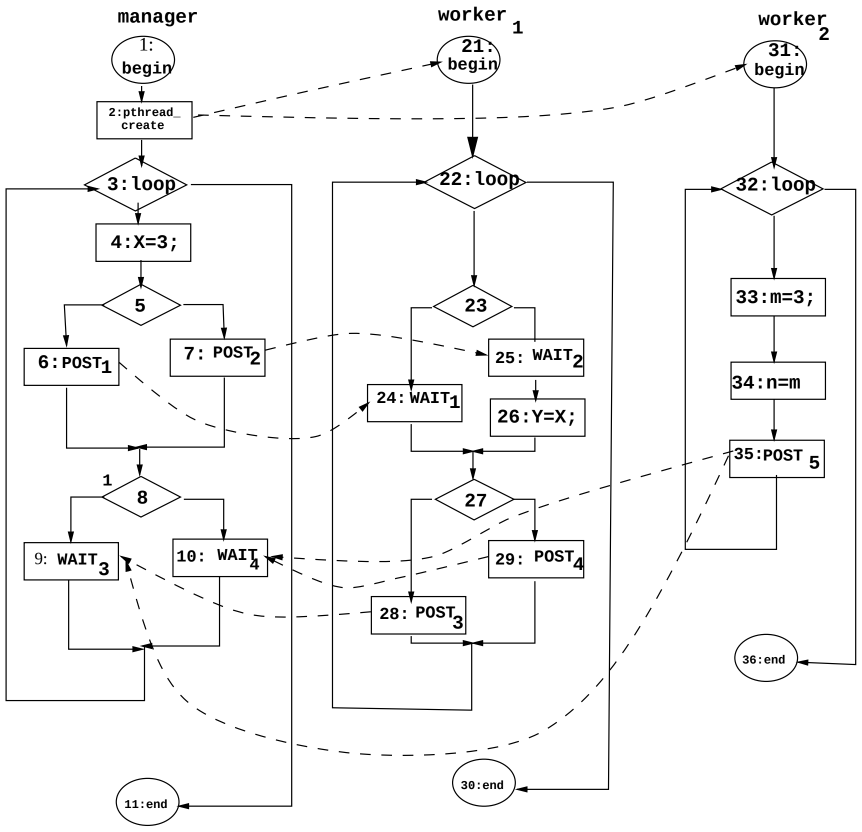


Figure 2: A More Complex PPFG Example

sequence 11-12-13-5-6, can be represented as $CS(11^2, 6^1)$. Also, in Figure 2, the connected sequence 4-5-7-25-26 can be represented as $CS(4^1, 26^2)$. There can be more than one connected sequence for a given pair of nodes. Since the sequence of nodes in a connected sequence does not associate with an executable sequence of statements because it is not a complete path, the sequence of nodes is called a *connection*.

**Definitions: Reach($\Rightarrow$) and Directly Reach($\Rightarrow_d$)**

Given a PPFG $G = (V,\ E)$ and two different nodes, $x^i$ and $y^j$, it is said that $x^i$ *reaches* $y^j$, denoted as $x^i \Rightarrow y^j$, if $\exists$ a connected sequence $CS(x^i,\ y^j)$, when $i \neq j$, or $\exists$ a path $P_i(x^i, y^j)$ when $i = j$. When $i = j$, it is said that $x$ *directly reaches* $y$, denoted as $x \Rightarrow_d y$.

In Figure 1, the *loop* node(10) in thread $T_2$ directly reaches the *post* node(13), i.e., *loop* $\Rightarrow_d$ *post*; and the $d$ node(11) in thread 2 reaches the $u$ node(6) in thread 1, i.e., $d \Rightarrow u$.

## 3 Structural Testing Framework for Shared Memory Programs

Nondeterminism complicates the process of testing parallel programs, making it difficult to establish a sound testing framework, with a reasonable expectation about the testing results. In the following, the all-uses criterion, a parallel testing paradigm, the subsumption hierarchy of structural testing data adequacy criteria are discussed and a testing procedure for dealing with the nondeterministic nature of parallel programs is described.

### 3.1 All-uses Criterion

In general, structural testing of a sequential program involves finding a path to cover statements in the program based on specific criteria, such as all-statements, all-branches, all-uses, all du-pairs, or all-paths. In this paper, the focus is on the all-uses criterion because the all-uses criterion subsumes both all-branches and all-statements criteria, but provides more reasonable testing cost effectiveness than more powerful criteria such as all-paths. The all-uses criterion is said to be satisfied by a test suite for a given program if the test suite ensures that for all variable definitions, each pair consisting of the definition and one of its reachable uses is covered by at least one definition-clear path for that variable[14].

In a sequential program, both the definition and the use of a variable are located in the same thread of execution. In contrast, the definition and the use of a shared variable in a shared memory parallel program can be located in two different threads. Thus, the notion of all-uses criterion must be modified.

**Definition: du-pair in a Parallel Program**

A *du-pair* in a parallel program is a triplet ( $var$, $n_u^i$, $n_v^j$ ), where $var$ represents a program variable defined in the statement represented by node $n_u^i$ and referenced by node $n_v^j$, and there is at least one path or connected sequence from $n_u^i$ to $n_v^j$ with no redefinition of $var$. The subscript $u$ represents the node id for the node $n_u^i$ in the unique numbering of the nodes in thread $T_i$; the subscript $v$ represents the node id for the node $n_v^j$ in the unique numbering of the nodes in thread $T_j$. If $n_u^i$ is a definition node and $n_v^j$ is a use node and $i = j$, the du-pair is called an *intraprocess du-pair*; if $i \neq j$, the du-pair is called an *interprocess du-pair*. To simplify the examples, a node $n_u^i$ will be simply written as $u^i$, such that a du-pair $(var, n_u^i, n_v^j)$ in the examples will be simply written as $(var, u^i, v^j)$.

In Figure 2, the variable $x$, defined in statement 4 and used in statement 26, is a shared variable. Statement 4 reaches statement 26 through a connected sequence 4-5-7-25-26 without redefinition of $x$. Hence, $(x, 4^1, 26^2)$ is a du-pair. In the thread called $worker_2$, the variable $m$ is a local variable. The triplet $(m, 33^3, 34^3)$ is an intraprocess du-pair since both statements 33 and 34 are located in the same thread.

**Definition: $node_1$ happens before $node_2$ ($node_1 \prec node_2$)**

In a parallel program, it is said that $node_1$ *happens before* $node_2$, or simply $node_1 \prec node_2$, if and only if there exists an instance of the node $node_1$ that completes execution before an instance of the node $node_2$ begins.

Some researchers have developed techniques for identifying the relation of *happened before*. For example, Grunwald and Srinivasan [15] developed a method to derive a conservative approximation to precedence information for parallel programs with *parallel sections* and *post* and *wait* synchronization features. Duesterwald and Soffa [10] developed a system of data flow equations to express a partial execution ordering in a *happened before* and a *happened after* relation for regions in Ada programs. In the rest of this paper, it is assumed that techniques are available for identifying the relation of *happens before.*

After a program is executed, a trace of the execution can reflect the execution sequence of every statement in a program. To reproduce the execution sequence of statements in a program, techniques such as deterministic execution(DET)[16] can be used. Reproducing an execution sequence of a path is often needed for debugging a program. In the replay, a path must be forced to be executed if a path is to be covered when the program being tested is executed. Formally, the term *force a path* is defined as follows.

**Definition: Forcing a Path**

A set of paths $PATH$ is said to be forced to execute, if for each *path* in $PATH$, for each node $n_1$ and its successor $n_2$ in *path*, $n_1 \prec n_2$ when the program being tested is executed.

**Definition: Path Coverage**

In thread $T_i$, node $n$ is *covered by a path* $P$, denoted by $n \in_p P$, if node $n$ occurs in the path $P$. The symbol $\in_p$ represents the *coverage* of a node. Node $n^l$ is considered to be *covered by a set of paths* $PATH$ $= (P_1, \ldots P_k)$ if $n^l \in_p P_l$ $(1 \leq l \leq k)$. This is denoted as $n^l \in_p PATH$.

**Definition: All-uses Test Case for a Parallel Program**

An *all-uses test case* for a given du-pair ( $var$, $n_u^i$, $n_v^j$ ) is defined to be a set of paths $PATH$, containing one path in each thread, such that $n_u^i \in_p PATH$, $n_v^j \in_p PATH$, $n_u^i \prec n_v^j$, and no other definition of *var* occurs between $n_u^i$ and $n_v^j$.

Conceptually, an all-uses test case must cover both the definition and the use nodes; there must be an instance of the definition node executed prior to an instance of the use node; no other definition of the variable defined at the definition node occurs between the definition node and the use node. In contrast to a single path in an all-uses test case for a sequential program, an all-uses test case for a parallel program is actually a set of paths, one path per thread of the program. As an example, an all-uses test case of the du-pair $(x, 4^1, 26^2)$ in Figure 2 is (1-2-3-4-5-7-8-9-3-11, 21-22-23-25-26-27-28-22-30, 31-32-33-34-35-32-36). The particular path in $worker_2$ is identified since node 9 is a *wait* node and the successful execution of a *wait* node requires the matching *post* in $worker_2$ to be executed.

## 3.2 Parallel Program Testing Paradigm

Generally speaking, *program testing* is an approach to control the flow of execution through a program in order to detect program faults. As a tester, one would test a program with different kinds of scenarios: good test cases and bad test cases. A program being tested should react (or work) successfully when 'good' test cases are used, and fail, terminate, or wait infinitely when 'bad' test cases are used.

Given a parallel program, one needs to test not only the execution flow of a program, but also the synchronization, communication, and thread creation events. Formally, a *test case* $TC$ is a 2-tuple ($\mathcal{PROG}$, $\mathcal{I}$) where $\mathcal{I}$ is the input data to the program $\mathcal{PROG}$, whereas a *temporal test case* $TTC$ is a 3-tuple ($\mathcal{PROG}$, $\mathcal{I}$, $\mathcal{D}$), where the third component, referred to as *timing changes*, is a parameter used for altering the execution time of program segments. While the timing of various events can be changed by explicitly varying scheduling decisions or inducing delays through heavy loading, another way is to introduce delay statements into the program[9], by code instrumentation. As part of this work, a static analysis was developed that first automatically, systematically and conservatively determines locations for delays in the program, and then automatically identifies unnecessary delay statements in order to systematically perform testing with timing changes, but also to reduce the number of testing runs[13].

Based on $\mathcal{D}$, the execution timing of certain synchronization instructions $n$, represented as $t(n)$, will be changed for each temporal test and the behavior of the program $\mathcal{PROG}$ will be observed. In practice, timing changes may be introduced progressively. Hence, the *level* of temporal testing is defined to be the total number of timing change statements, e.g., sleep(duration), introduced in each test case.

Conceptually, given a parallel program $\mathcal{PROG}$, one must execute $\mathcal{PROG}$ first without and then with timing changes. That is, for each input, the program is executed many times (at least more than once) based on a specific testing criterion $\tau_l$, where $l$ is the *level* of temporal testing. The *execution result* of the shared memory parallel program $\mathcal{PROG}$ using the test case $t$ is the set of values of all shared variables in the program $\mathcal{PROG}$ when the program terminates. If all test runs show the same execution result, this set of test data is considered $\tau_l$ adequate. If no timing changes are used, the test suite is $\tau_0$ adequate. If the level of temporal testing is $i$, this test suite is called $\tau_i$ adequate. That is, if a parallel program is executed many times with a test suite and all trace results indicate that the same paths are covered, the test suite is considered $\tau_l$ adequate. For example, given a du-pair, a test suite to cover the du-pair is generated. Then each path is tested without any delay. If the same paths are covered when the program is executed many times for a du-pair, the test suite is considered $\tau_0$ adequate with respect to that du-pair. There is no general rule for how many times a program should be executed in each test case. Also, the problem of finding all locations that require timing changes in polynomial time is still an open problem, similar to the issue of finding all paths in any program; it is not guaranteed that all places that require timing changes can be located in polynomial time.

## 3.3 Structural Testing Coverage Criteria

Intuitively, a *testing criterion* defines what needs to be tested. It can also be considered as a criterion for checking whether or not a test suite is adequate for applying the criterion to a program. Thus, formally a *test data adequacy criterion* has been defined in other literature [17] as follows.

*A test suite T is C adequate for a given sequential program P if there exist some test cases in T, such that the test data adequacy criterion C is satisfied (by the test cases) .*

Each test data adequacy criterion is based on covering some entity of programs. For example, the statement adequacy criterion involves covering each statement in a program; the data flow adequacy criterion involves covering du-pairs; and path testing criteria involve covering paths in a program. The symbol $COVERED_C(P, t)$ is used to represent the set of covered entities in $P$ associated with the criterion $C$ when the program $P$ is executed using the test case $t$. A test data adequacy criterion is *satisfied* by a test suite T if every entity associated with the criterion in the program is covered by a test case $t$ in T. For example, for a given program PPFG=$(V, E)$, the test data adequacy criterion *all statements* is *satisfied* by a set of test cases $T$ if $\forall v \in V, \exists t \in T,$ *such that* $v \in COVERED_C(P, t)$.

In order to extend the notion of test coverage criteria to the world of parallel programs, the following terms are defined: *consistent* and *strictly consistent* execution results. The *execution result* for the $i$th execution of such a program is represented by $\mathcal{R}^P(t; i)$. Two execution results for the $i$th and $j$th execution runs are *consistent* if $\mathcal{R}^P(t; i) = \mathcal{R}^P(t; j)$. If for all $i$, $\mathcal{R}^P(t; i)$ stays unchanged, the execution results of program $P$ are *strictly consistent*. Obviously, it is impractical to try to show that all possible execution results of a program are strictly consistent. It can only be expected to show that no inconsistent execution results are produced after a program is executed many times using the same input. This term is only introduced for completeness.

With this background, *weak test data adequacy criterion* and *test data adequacy criterion* for parallel programs are defined as follows.

*For a program $P$, a test suite $T$ is weakly $C$ adequate if there exists a test suite $T$ such that the test data adequacy criterion $C$ is satisfied by $T$ after $P$ is executed more than once using a test case $t$ in $T$, and execution results are always consistent. If for all possible execution runs, the results are strictly consistent, $T$ is $C$ adequate.*

If a program $P_2$ is transformed from another program $P_1$ by inserting statements such as `sleep()` into $P_1$ for changing the execution timing of some statements in $P_1$, then the program $P_2$ is called a *mutant program*. The concepts of *weak test data adequacy criterion with timing changes* and *test data adequacy criterion with timing changes* are defined for parallel programs as follows.

*For a program $P$, a test suite $T$ is weakly $C$ adequate with timing changes if there exists a mutant program $P_m$ of $P$ such that $C$ is satisfied by $T$ after $P_m$ is executed more than once using a test case $t$ in $T$, and consistent execution results are always produced. If $T$ is weakly $C$ adequate with timing changes and all*

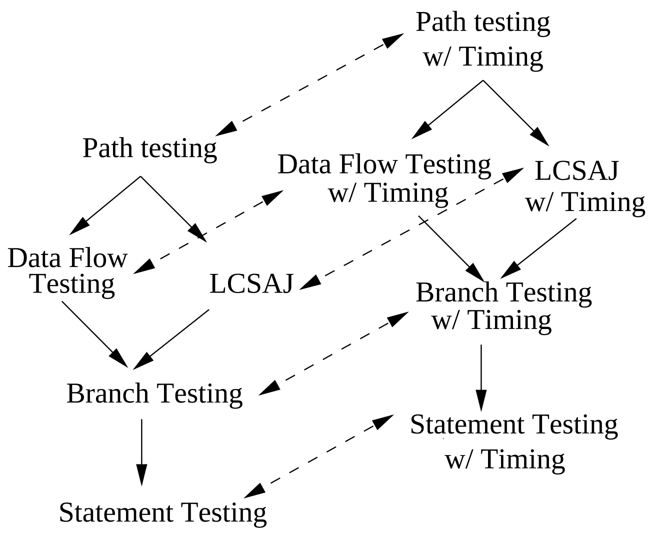


Figure 3: Testing Criteria Subsumption Hierarchy for Parallel Programs

*possible execution results of the mutant program $P_m$ are strictly consistent, then $T$ is $C$ adequate with timing changes for $P$.*

Although test data adequacy criteria are used individually, they are not independent of each other. A set of test data that is adequate with respect to one criterion sometimes implies that it is adequate for another criterion. This relation between test data adequacy for parallel programs is explored in the next section.

## 3.4 Nondeterminism and the Testing Process

Based on the definitions in the previous section, a subsumption hierarchy for parallel programs can be established, similar to the notion of a subsumption hierarchy for test adequacy criteria for sequential programs.

The concept of *subsumption* has been defined previously in the literature [17] as follows.

*Given a program $P$ and two criteria $A$ and $B$, it is said that $A$ subsumes $B$ if when a test suite $T$*

*is A adequate then T is also B adequate.*

The concept of *weak subsumption* is formally introduced as follows.

*Given a program P and two criteria A and B, it is said that A weakly subsumes B if when a test suite T is weakly A adequate (with or without timing changes) then T is also weakly B adequate.*

Figure 3 shows the subsumption hierarchy for parallel programs. Conceptually, path testing generates test data to cause the execution of all paths, one in each equivalence class of paths in a program. For the purpose of reducing test cases, the test case for iterating a loop in a program once and the test cases for iterating the loop more than once can be considered to be in the same equivalence class. A *linear code sequence and jump* (LCSAJ) is a 3-tuple $(s_1, s_2, s_3)$, where $s_1$ is the first statement of a program or the target statement of a control flow jump, $s_2$ is the ending statement, (i.e., any statement which can be reached from the start statement by a consecutive sequence of code and from which a jump can be made,) and $s_3$ is the target statement to which execution will jump after the end statement $s_2$ is executed. This criterion requires that all LCSAJs be executed. All-uses is one of the data flow testing criteria.

In Figure 3, the subsumption relations between the criteria of path testing are shown with solid arrows. Weak subsumption relations are shown with dotted arrows. The temporal testing criteria hierarchy is similar to the hierarchy representing the subsumption relations among testing criteria without timing changes.

It is claimed without proof that a temporal testing criterion with timing changes weakly subsumes a corresponding testing criterion without timing changes, and vice versa. For example, all-uses (shown as data flow testing in the figure) coverage with timing changes weakly subsumes all-uses coverage without timing changes. The reason is obvious. Without the existence of synchronization errors, consistent results will be produced no matter how synchronization events are delayed. For example, if $T$ is weakly *all statements* adequate for $P$ with timing changes, $T$ is also weakly *all statements* adequate for $P$ without timing changes, and vice versa. However, this could not be true with the presence of synchronization errors. When synchronization errors exist, a temporal testing criterion could be satisfied when executing a mutant program with timing changes using a test suite $T$, but could fail to be satisfied when the program is executed without timing changes using $T$, and vice versa. For example, if $T$ is *all statements* adequate for $P$ with timing changes, $T$ could fail to be *all statements* adequate without timing changes.

## 4 All-uses Testing

This section describes the problem of finding test cases for all-uses testing, discusses the issues faced in finding all-uses test cases for a given du-pair in a parallel program using some examples, and then defines and characterizes a classification of all-uses test cases in parallel programs.

### 4.1 Problem Definition

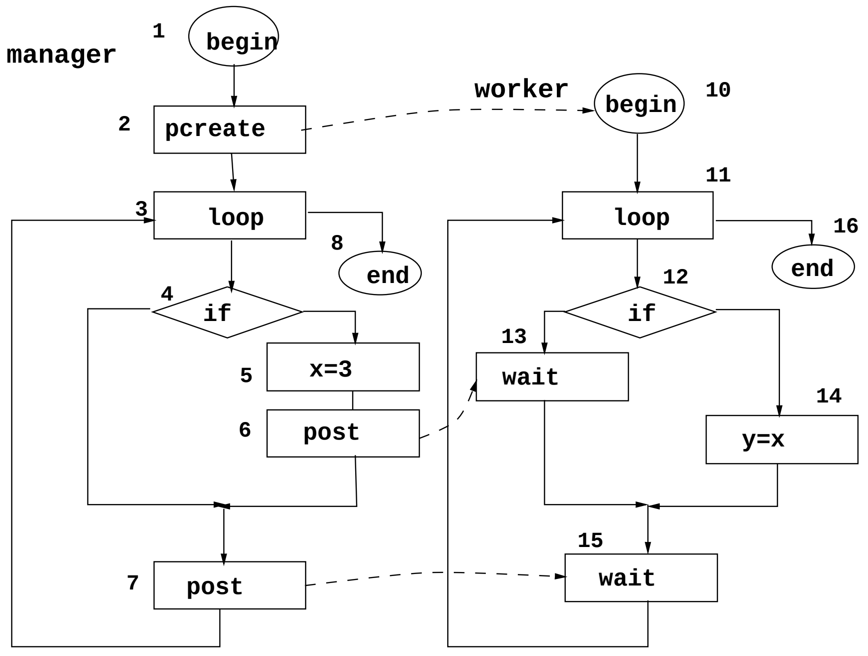
The problem of finding all-uses test cases for testing a shared memory parallel program can be stated as follows:

Given a shared memory parallel program, $\mathcal{P}ROG = (T_1, T_2, \ldots, T_n)$, for each du-pair, ( $var$, $n_u^i$, $n_v^j$ ), in $\mathcal{P}ROG$, find a set of paths $PATH = (P_1, \ldots P_k)$ in threads $T_1$, $T_2, \ldots T_k$, respectively (one path per thread) that covers the du-pair ( $var$, $n_u^i$, $n_v^j$ ), such that $n_u^i \prec n_v^j$.

One of the objectives of identifying test cases that provide all-uses coverage is to be able to generate "valid" test cases for testing the normal behavior of shared memory parallel programs adequately. Invalid test cases can be either infeasible or incorrect, i.e., one that violates the definition of the all-uses criterion. Although some "invalid" test cases can also be used for testing the behavior of a program under abnormal situations, this paper only focuses on generating valid test cases for all-uses testing.

### 4.2 Inherent Problem Exhibited

In this section, simplified examples are used to demonstrate why invalid all-uses test cases can be generated in a parallel program with synchronization features. An invalid test case can be caused by including an inconsistent number of loop iterations in related threads that are connected by synchronization edges, an incorrect sequence of synchronization nodes in related threads, or by choosing an ”incorrect” branch at a loop node. The issues illustrated here are not necessarily exhaustive, but instead meant to illustrate the complexity of the problem of automatically identifying an all-uses test case for parallel programs. In addition to these issues, a sound algorithm must be capable of finding a complete path in each thread. Direct application of the path finding algorithms for sequential programs would find partial paths, not complete paths, in parallel programs.



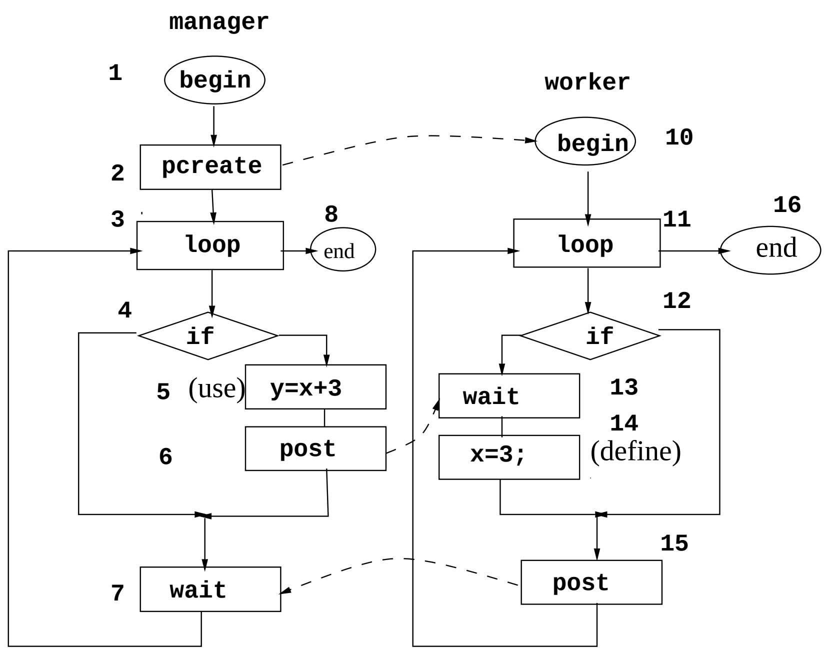
**PATH COVERAGE:**

**MANAGER:1-2-3-4-5-6-7-3-8**

**WORKER:10-11-12-13-15-11-12-14-15-11-16**

Figure 4: An example of an invalid all-uses test case

Figure 4 contains two threads, the `manager` thread and a `worker` thread. This figure demonstrates an invalid all-uses test case that indeed covers the du-pair, but does not cover both the *post* and the *wait* of a matching *post* and *wait*. In a valid all-uses test case, for each *wait*, at least one matching *post* should be included in all threads.



**PATH COVERAGE:**
**MANAGER:1-2-3-4-5-6-7-3-8**
**WORKER:10-11-12-13-14-15-11-16**

Figure 5: Another example of an invalid all-uses test case

Figure 5 illustrates another example of an invalid all-uses test case. The generated paths cover both the *define* of $x$ at node 14 and its *use* at node 5, but the *use* node will be reached before the *define* node, that is, $define \not\prec use$. 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 `manager` thread, then an attempt to find a path coverage that will test this du-pair should be made. The invalid path coverage does not provide this coverage.

In Figure 5, if the generated paths are 1-2-3-4-7-3-8 in the `manager` thread and 10-11-12-13-14-15-11-16 in the `worker` thread, then the all-uses test case is invalid due to the choice of an incorrect branch (edge 4-7 instead of edge 4-5) in the `manager` thread. Hence, the selection of a branch at an *if*-node is also important.

## 4.3 A Classification of All-uses Test Cases for Parallel Programs

The previous examples motivate a classification of all-uses test cases. In addition to classifying an all-uses test case to be valid or invalid, an all-uses test case can be classified as either *acceptable* or *unacceptable*, and as either *w-runnable* or *non-w-runnable.* Conceptually, an all-uses test case $PATH$ is *acceptable* if the definition and the use nodes are both covered in the order such that the definition node happens before a post node, a post happens before a wait node, and a wait node happens before the use node. Furthermore, an acceptable $PATH$ is *w-runnable* if the set of paths $PATH$ can be used to generate a test case that does not cause an infinite wait in any thread. To achieve this, for each instance of a wait in $PATH$, $PATH$ must also cover an instance of a matching post.

For the convenience of explanation, the concept of *matching nodes* is defined. The set of matching *post* nodes of a *wait* node $w$ is the set $MP$, where $MP(w) = \{p|(p,w) \in E_s\}$ The set of matching *wait* nodes of a *post* node $p$ is the set $MW$, where $MW(p) = \{w|(p,w) \in E_s\}$

### 4.3.1 Acceptability of All-Uses Coverage

A set of paths $PATH$ is *acceptable* based on the all-uses criterion, or $PATH_a$, for the du-pair ($var$, $define$, $use$) in a parallel program free of infeasible paths if all of the following conditions are satisfied:

1. $define \in_p PATH$; $use \in_p PATH$,
2. $\forall$ *wait* nodes $w \in_p PATH$, $\exists$ a *post* node $p \in MP(w)$, such that $p \in_p PATH$,
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 set of paths $PATH$ (condition 1); for each *wait* node, at least one matching *post* node is covered by $PATH$ (condition 2); any *(post,wait)* edges 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 (condition 3); and for each head of a thread creation edge $e$, i.e., $H(e)$, the associated tail of $e$, $T(e)$, is included in the path(condition 4). If any of these conditions is not satisfied, the set of paths is considered to be *unacceptable.* For example, if only a *wait* is covered in an all-uses test case and none of the matching *post* nodes are covered, the test case is not a $PATH_a$. Figure 5, where the define and use are covered in reverse order, shows an instance that only satisfies the first two and last conditions, but fails to satisfy the third condition. Thus, the paths shown in Figure 5 do not achieve acceptable all-uses coverage.

### 4.3.2 W-runnability of All-uses Coverage

Section 4.2 illustrates that a parallel program can cause infinite wait, even when the test case is acceptable. More formally, a set of paths is w-runnable if it is a $PATH_a$ and all of the following additional conditions are satisfied. In the remainder of this paper, unless specified explicitly, the subscript $i$ of a node $n_i$ is used to represent an instance of the node $n$.

1. For each instance $i$ of a *wait* $w^t$, $w^t_i \in_p PATH$, (possibly represented by the same node $n^t \in_p PATH$), $\exists$ an instance $u$ of a *post*, $p^s_u \in_p PATH$, where $p^s_u \in MP(w^t_i)$. 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.
2. The execution of the paths in $PATH$ will not cause a circular wait. That is, for a program with threads $T_0$, $T_1$, ..., and $T_n$, $\nexists$ threads $T_{i_0}$, $T_{i_1}$, ..., $T_{i_m}$, such that $(w^{i_0} \prec p^{i_0} \wedge w^{i_1} \prec p^{i_1} \wedge \ldots \wedge w^{i_m} \prec p^{i_m})$ $\wedge$ $(p^{i_0} \prec w^{i_1} \wedge p^{i_1} \prec w^{i_2} \wedge \ldots \wedge p^{i_{m-1}} \prec w^{i_m})$ $\wedge$ $(p^{i_m} \prec w^{i_0})$, where $1 \leq m \leq n$, $0 \leq i_x \leq n$, $\forall\, x$, $0 \leq x \leq m$.

The first condition above ensures that, for each instance of a *wait* in the $PATH$, there is a matching instance of a *post*. However, it is not required that for every instance of a *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 $w \in_p PATH$. The second condition above ensures that a deadlock will not occur due to a circular wait. For example, for a program with two threads $T_i$ and $T_j$, $\nexists$*post* nodes $p^i$, $p^j$, and *wait* nodes $w^i$, $w^j$, $\in_p PATH$ such that $(p^i \prec w^j) \wedge (p^j \prec w^i) \wedge (w^i \prec p^i) \wedge (w^j \prec p^j)$. However, these conditions do not guarantee the completion of execution; the classification is solely based on the semantics of synchronizations associated with *post* and *wait* but not *all* program statements.

Although all $PATH_w$ test cases can also be classified as $PATH_a$, the term $PATH_a$ and non-$PATH_a$ will be used to strictly refer to non-$PATH_w$ $PATH_a$ and non-$PATH_w$ non-$PATH_a$ test cases, respectively. Figure 4 shows an example of a $PATH_a$ test case due to the fact that the first necessary condition for a $PATH_w$ is violated. Figure 5 shows a non-$PATH_a$ for which the third necessary condition for a $PATH_a$ is violated.

From the viewpoint of software testing, it is equally desirable to find all $PATH_w$ and all $PATH_a$ test cases, respectively, because to test a program, one would use test cases for which successful results are expected as well as test cases for which the program should report an error. In other words, program testing should target both types of program behavior: (1) a program fails when successful results are expected, and (2) a program terminates successfully without reporting synchronization errors when in fact there are problems that should be reported. Hence, $PATH_w$ test cases will be used to test the first type of program behavior, whereas $PATH_a$ will be used to test the second type of program behavior.

## 5 All-uses Testing of Parallel Programs in Practice

### 5.1 Computational Complexity Issues of All-uses Testing

Several computational complexity issues arise with respect to the all-uses criterion for parallel programs. In particular, here are three important questions to be answered in order to direct testing efforts appropriately.

1. $EXIST_a$ - Given a PPFG G=(N,E) and a du-pair $(var, d, u)$, does a $PATH_a$ exist in G to cover $(var, d, u)$?

2. $CLASSIFY\_PATH$ - Given a PPFG G=(N,E) and a path coverage $PATH$, is $PATH$ a $PATH_w$?

3. $EXIST_w$ - Given a PPFG G=(N,E) and a du-pair $(var, d, u)$, does a $PATH_w$ exist in G to cover $(var, d, u)$?

In the Ph.D. dissertation of Yang [13], proofs of the following three important theorems are described in detail. For a given parallel program $\mathcal{PROG}$ and a du-pair, it is shown that

1. $EXIST_a$ is in **P**, and
2. $CLASSIFY\_PATH$ is in **P**,
3. $EXIST_w$ is **NP-C**.

The first theorem is proved by presenting an algorithm that is capable of finding a $PATH_a$ in deterministically polynomial time when a $PATH_a$ exists in a program. The second theorem is shown by presenting a polynomial time algorithm to determine the classification of an all-uses test case. The third theorem is shown by linearly reducing the 3-$SATISFIABILITY$ problem, which is a known problem in **NP-C**, to the problem of $EXIST_w$. From these results, one should only expect to find $PATH_a$ in polynomial time since determining whether or not a $PATH_w$ exists is **NP-C**. Hence, this research focuses on finding $PATH_a$ test cases for a given du-pair.

## 5.2 Empirical Studies

Experimental studies have been conducted by many researchers to investigate the fault detecting capabilities of structural testing methodologies in the context of sequential programs [18, 19, 20, 21, 22]. The effectiveness of temporal testing methodologies has also been studied in the context of concurrent programs [23]. The major goal of the experimental studies described in this paper is to understand the effectiveness of the algorithms designed and implemented in [13] for finding an all-uses test case in a shared menory parallel program as well as the characterization of all-uses test cases in real programs. In particular, the empirical studies are designed to investigate the following questions:

1. For a given du-pair, what program characteristics cause a particular test case to be classified as $PATH_a$?

2. When the test case generation algorithm given in [13] returns a test case of the type $PATH_a$, are there $PATH_w$ cases for the given du-pair that the algorithm did not find?

3. When a $PATH_w$ test case is found, are there $PATH_a$ test cases for the given du-pair, i.e., does the algorithm find a $PATH_w$ when in fact $PATH_a$ test cases exist?

The answer to question one gives insight into possible causes for synchronization errors in a shared memory parallel program. Questions two and three provide clues on what test suites should be expected to be generated when the problem of determining whether or not a $PATH_w$ exists has been proven to be NP-C.

Three *representative* programs were chosen for conducting the experimental study: a producer/consumer program, a pipelining program, and a TCP and UDP name caching service program [24]. These three programs vary in size, number of synchronization calls, and number of condition variables. Although the number of programs used in the experimental studies is small, these programs represent typical applications of using the mutual exclusion locks and the *post/wait* synchronization operations for controlling the execution sequence of synchronization events.

### 5.2.1 Producer/Consumer

The producer/consumer program generates three worker threads from the producer thread. The producer thread and one worker thread jointly work as a "team" and another two worker threads as another "team" for processing input data. The input data will be randomly received by one of the two producer threads, and the consumer thread associated with the producer thread will then perform computation activities using the data. The producer/consumer program structure has been found extensively in many shared memory parallel applications. For example, a client/server paradigm is a producer/consumer problem. Even the paradigm of software pipelining can be considered another example of the producer/consumer problem. In general, both data parallelism and function parallelism can be achieved using the producer/consumer paradigm. Data parallelism can be achieved with one producer and multiple consumers, whereas function parallelism can be

```
 thread 1:                          thread 2:

...                                ...
while (full_buf == 0)              if (full_buf ==0)
   wait(some_full, &lock)             post(some_full);
...                                ...
```

Figure 6: Pattern of a Matching *post* And *wait* Causing *non-$PATH_w$*

achieved with equal number of producers and consumers. Pipelining is simply a special combination of data parallelism and function parallelism [24].

In the producer/consumer program, the algorithm for generating an all-uses test case returns 19 (or 54.3%) $PATH_w$ and 16 (or 45.7%)$PATH_a$ test cases in total with respect to 35 du-pairs. In this benchmark program, the reason that a test case becomes $PATH_a$ is mainly due to the coding style of several *wait* events. For example, Figure 6 illustrates one matching *post* and *wait* event in this program. If the *wait* is executed twice and the *post* is only executed once, the test case becomes non-$PATH_w$. In this case, the all-uses test case generation algorithm reports a $PATH_a$ test case which could potentially cause a possible program fault, while there are indeed no coding errors. In general, if a $PATH_a$ test case is found, a synchronization error might exist in the program. However, in this case, the empirical result indeed illustrates a possible "false alarm." Although this programming style caused a non-$PATH_w$ test case to be reported in this program, this is definitely not the only possible cause of the test case finding algorithm to find a non-$PATH_w$ test case. Resolving this situation will not guarantee that the algorithm finds $PATH_w$ test cases in all programs.

Ideally, one would like the path finding algorithm to find the expected $PATH_w$ test cases when they exist. However, the effort to find a $PATH_w$, if one exists, greatly increases the complexity of the path finding algorithm due to the need to backtrack on paths already found.

### 5.2.2 Pipelining

A simplified software pipelining program with three stages in this pipeline is studied. Each stage is represented and performed by one thread. Many applications use software pipelining to achieve higher parallelism.

For example, protocol processing on a host machine has been proposed to achieve higher performance using software pipelining [25, 26, 27]. The result of the all-uses path finding algorithm indicate that 2 (or 25%) $PATH_a$ test cases and 6 (or 75%) $PATH_w$ test cases are found. The $PATH_a$ test cases are generated due to a situation similar to the producer/consumer program.

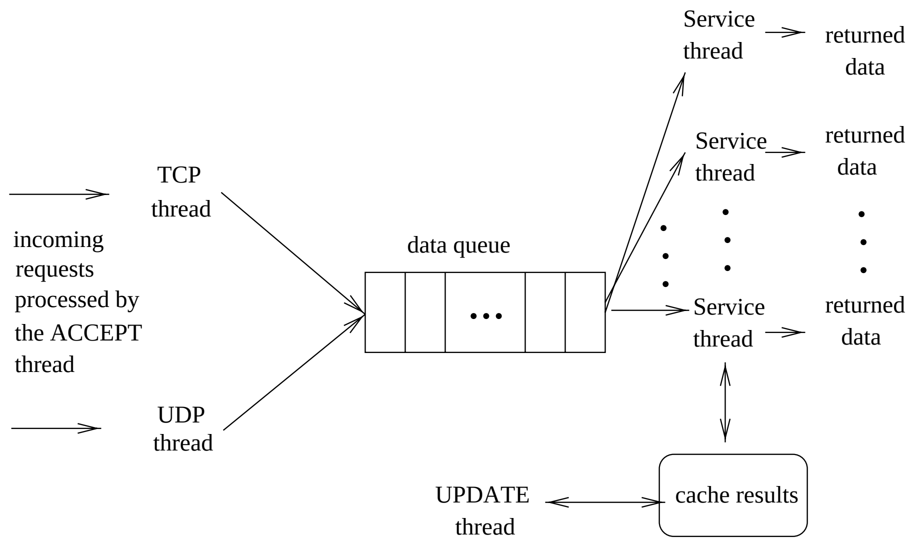


Figure 7: A TCP and UDP Name Caching Program

### 5.2.3 TCP and UDP Name Caching Service

Figure 7 illustrates the program structure of the TCP and UDP name caching service. When TCP or UDP protocol processing takes place, an IP address, such as "128.175.1.9" is used to identify a location, i.e., a host, in a network; a domain name such as "cis.udel.edu" must be translated into an IP address for the purpose of processing. Thus, domain names used once are usually stored in a high speed cache memory so that the speed of translating the same domain name can be expedited. In general, the name cache service provided by the TCP or UDP protocol server is usually provided by running a name caching system as a daemon, which is simply a process that will never terminate as long as the machine is up and running [24]. The name caching service is implemented as a multithreaded daemon for caching the results of service requests associated with many host names. In total, five threads are created by the manager thread including TCP, UDP, ACCEPT, SERVICE, and UPDATE thread. Various SERVICE threads use a single data queue to order service requests. The SERVICE thread performs all the lookup, update, and delete functionalities. The UPDATE thread periodically scans the hash table and refreshes the most commonly used host names and discards any whose time-to-live, i.e., the life-time of a data item, has expired. This program is more complicated than the other two benchmark programs in its number of *post* and *wait* statements, and the number of condition variables.

The experimental results of finding all-uses test cases show similar behavior to the other two representative programs. Of the 32 test cases, 22 (or 68.75%) are $PATH_w$, 10 (or 31.25%) are $PATH_a$. That is, some reported test cases are $PATH_a$ due to the fact that a *wait* statement is included twice and the matching *post* statement is included only once in the $PATH_a$ test case. For some cases, there exist $PATH_w$ when a $PATH_a$ test case is reported, and vice versa. The reasons are explained in previous sections.

### 5.2.4 Summary of Results

| Program | $PATH_w$ | $PATH_a$ |
|---|---|---|
| Producer/Consumer | 54.3% | 45.7% |
| Pipelining | 75% | 25% |
| Name Caching | 68.75% | 31.25% |

Table 1: Results of the Path Finding Algorithm

In summary, Table 1 lists the percentages of $PATH_w$ and $PATH_a$ paths, respectively, found in each of the three programs. Some conclusions can be made from conducting these experiments. They are described as follows:

1. For a given du-pair, synchronization errors can cause a $PATH_a$ to be generated. However, some coding styles can also cause $PATH_a$ to be generated. An example of such a coding style is given in Figure 6.

2. The all-uses test case generation algorithm in [13] may find a $PATH_a$ when a $PATH_w$ test case exists. This result is consistent with the theoretical result. Since it has been shown that determining whether or not a $PATH_w$ test case exists for covering a du-pair is **NP-C**, the polynomial time algorithm to find $PATH_a$ test cases is developed even when $PATH_w$ may exist. Ideally, when $PATH_w$ test cases

exist, a polynomial time algorithm should be able to produce a $PATH_w$ test case as specified by a user. However, unless $\mathbf{P} = \mathbf{NP}$, it is impossible. One should only expect to find $PATH_a$ test cases and manually modify the paths if necessary.

3. The all-uses test case generation algorithm can report a $PATH_w$, if one exists, when a (non-$PATH_w$) $PATH_a$ may also exist. The fact that a $PATH_w$ is returned does not guarantee that another $PATH_a$ does not exist. It is certainly desirable to be able to find as many $PATH_a$ cases as possible so that more possible synchronization errors can be detected. However, this issue is beyond the scope of this paper. The algorithm in [13] only expects to be able to find a $PATH_a$ in polynomial time.

## 6 Related Work

In the context of sequential programs, several researchers have examined the problems of generating test cases using path finding as well as finding minimum path coverage [28, 29, 3]. All of these methods for finding actual paths focus on programs without parallel programming features and, therefore, cannot be applied directly to finding all-uses coverage for parallel programs. However, the depth-first search approach and the approach of using dominator and post-dominator trees can be used together with extension to provide all-uses coverage for parallel programs. The hybrid method for finding all-uses test cases has been published in [30, 31].

Gabow, Maheshwari, and Osterweil [28] showed how to use depth-first search (DFS) to find actual paths that connect two nodes in a sequential program. When applying DFS alone to parallel programs, it is not appropriate even for finding $PATH_a$, not to mention $PATH_w$. The reason is that although DFS can be applied to find a set of paths for covering a du-pair, this approach does not cope well with providing coverage for any intervening *wait*'s, and the corresponding coverage of their matching *post*'s as required to find $PATH_a$. For example, consider a situation where there are more *wait* nodes to be included while completing the partial path for covering the *use* node. Since the first path is completed and a matching *post* is not included in the original path, the first path must be modified to include the *post*. This is not a straightforward task, and becomes a downfall of using DFS in isolation for providing all-uses coverage for parallel programs.

Bertolino and Marrè have developed an algorithm (which is called DT-IT in this paper) that uses dominator trees (DT) and implied trees (IT) (i.e., post-dominator trees) to find path coverage for all branches in a sequential program [3]. A dominator tree is a tree that represents the dominator relationship between nodes (or edges) in a control flow graph, where a node $n$ dominates a node $m$ in a control flow graph if every path from the entry node of the control flow graph to $m$ must pass through $n$. Similarly, a node $m$ postdominates a node $p$ if every path from $p$ to the exit node of the control flow graph passes through $m$.

The DT-IT approach finds all-branches coverage for sequential programs as follows. First, a DT and an IT are built for each sequential program. Edges in the intersection of the set of all leaves in DT and IT, defined as *unconstrained edges*, are used to find the minimum path coverage based on the claim that if all unconstrained edges are covered by at least one path, all edges are covered. The algorithm finds one path to cover each unconstrained edge. When one edge is selected, one sub-path is found in DT as well as in IT. When one node and its parent node in DT or IT are not adjacent to each other in the control flow graph of the program, users are allowed to define their own criteria for connecting these two nodes to make the path. The two sub-paths, one built using DT and the other built using IT, are then concatenated together to derive the final path coverage.

If an attempt to run this algorithm is made to find all-uses coverage for parallel programs, generating test cases for the all-uses criterion instead of all-edges is needed, which is a minor modification. However, this approach will also run into the same problem as in DFS. That is, if some *post* or *wait* is reached when a path is being completed, the path must be adjusted to include the matching nodes. In addition, another problem is created regarding the order in which the *define* and *use* nodes are covered in the final path. Thus, using this method alone cannot guarantee detection of a $PATH_a$.

Yang and Chung [32] proposed a model to represent the execution behavior of a concurrent program, and described a test execution strategy, testing process and a formal analysis of the effectiveness of applying path analysis to detect various faults in a concurrent program. An execution is viewed as involving a concurrent path (C-path), which contains the flow graph paths of all concurrent tasks. The synchronizations of the tasks are modelled as a concurrent route (C-route) to traverse the concurrent path in the execution, by building a rendezvous graph to represent the possible rendezvous conditions. The testing process examines the correctness of each concurrent route along all concurrent paths of concurrent programs. Their paper acknowledges the difficulty of C-path generation; however, the actual methodologies for the selection of C-paths and C-routes are not presented in the paper.

## 7 Conclusions and Future Directions

The main focus of this paper has been to examine a methodology by which structural testing criteria for sequential programs can be extended to shared memory parallel programs. Foundations for structural testing of parallel programs have been laid, with formulation of the all-uses criterion for parallel programs, a testing framework that addresses nondeterminism, classifications of all-uses test cases, and theoretical results that indicate the expectations one can have for all-uses testing of parallel programs.

To the authors' knowledge, this is the first effort to extend the notion of all-uses coverage to shared memory parallel programs. In separate papers, an algorithm for identifying all-uses test cases for parallel programs [30, 31] is described in detail.

Future work includes extensions to the methodology to handle more language constructs, such as *parallel do*, *barrier*, and *clear*. An extension of this work to cover def-use pairs that involve procedure calls along the paths by applying interprocedural program representations is being investigated. To effectively deal with source code changes, incremental or regression testing becomes an important area for further investigation. The generation of structural testing data is generally an expensive task. An incremental test case generation approach could possibly save the computational cost of generating a full test suite again.

## References

[1] M. R. Girgis and M. R. Woodward. An integrated system for program testing using weak mutation and data flow analysis.

In *International Conference on Software Engineering*, pages 313–319, Imperial College, London, UK, 1985.

[2] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[3] Antonia Bertolino and Martina Marrè. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, 20(12):885–899, December 1994.

[4] Shing Chi Cheung and Jeff Kramer. Tractable dataflow analysis for distributed systems. *IEEE Transactions on Software Engineering*, 20(8):579–593, August 1994.

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

[6] Michal Young, Richard N. Tylor, Kari Forester, and Debra Brodbeck. Integrated concurrent analysis in a software development environment. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 200–209, Key West, Florida, December 1989.

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

[8] 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.

[9] Sergio Silva. *Mutation Analysis of Concurrent Software*. PhD thesis, Politecnico di Milano, Italy, January 1998.

[10] Evelyn Duesterwald and Mary Lou Soffa. Concurrent analysis in the presence of procedures using a data-flow framework. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 36–48, Vancouver, Canada, October 1991.

[11] Vasanth Balasundaram and Ken Kennedy. Compile-time detection of race conditions in a parallel program. In *1989 International Conference on Supercomputing*, pages 175–185, Heraklion, Crete, Greece, June 1989.

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

[13] Cheer-Sun David Yang. *Program-based, structural testing of shared memory parallel programs*. PhD thesis, University of Delaware, August 1999.

[14] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[15] 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.

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

[17] E. J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):668–675, June 1988.

[18] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[19] Elain J. Weyuker. More experience with data flow testing. *IEEE Transactions on Software Engineering*, 19(9):912–919, September 1993.

[20] Joe W. Duran and Simon C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, July 1984.

[21] Phyllis G. Frankl and Elaine J. Weyuker. Provable improvements on branch testing. *IEEE Transactions on Software Engineering*, 19(10):962–975, October 1993.

[22] Phyllis G. Frankl and Elaine J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202–213, March 1993.

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

[24] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with Threads.* SunSoft Press, 1996.

[25] Ahmed N. Tantawy. *High Performance Networks.* Kluwer Academic Publishers, 1994.

[26] David D. Clark and David L. Tennenhouse. Architecture considerations for a new generation of protocols. In *ACM SIGCOMM*, pages 200–208, Philadelphia, PA,USA, 1990.

[27] Niraj Jain, Mischa Schwartz, and Theodore R. Bashkow. Transport protocol processing at gbps rates. In *ACM SIGCOMM*, pages 188–199, Philadelphia, PA,USA, 1990.

[28] H. N. Gabow, S. N. Maheshwari, and L. J. Osterweil. On two problems in the generation of program test paths. *IEEE Transactions on Software Engineering*, SE-2(3):227–231, September 1976.

[29] S. C. Ntafos and S. Louis Hakimi. On path cover problems in digraphs and applications to program testing. *IEEE Transactions on Software Engineering*, 5(5):520–529, September 1979.

[30] C.-S. D. Yang and L. L. Pollock. An algorithm for all-du-path testing coverage of shared memory parallel programs. In *Sixth Asian Test Symposium*, pages 263–268, Akita, Japan, November 1997.

[31] Cheer-Sun D. Yang, Amie L. Souter, and Lori L. Pollock. All-du-path coverage for parallel programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 153–162, Clearwater Beach, FL, March 1998.

[32] R-D Yang and C-G Chung. A path analysis approach to concurrent program testing. In *9th Annual Phoenix Conference on Computers and Communications*, pages 425–432, Phoenix, AZ, 1990.