All-du-path Coverage for Parallel Programs

Cheer-Sun D. Yang Amie L. Souter Lori L. Pollock

Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716
(302) 831-1953 (302) 831-8458 (Fax)
{yang,souter,pollock}@cis.udel.edu

Abstract

One significant challenge in bringing the power of parallel machines to application programmers is providing them with a suite of software tools similar to the tools that sequential programmers currently utilize. In particular, automatic or semi-automatic testing tools for parallel programs are lacking. This paper describes our work in automatic generation of all-du-paths for testing parallel programs. Our goal is to demonstrate that, with some extension, sequential test data adequacy criteria are still applicable to parallel program testing. The concepts and algorithms in this paper have been incorporated as the foundation of our DELaware PArallel Software Testing Aid, **della pasta**.

Keywords: parallel programming, testing tool, all-du-path coverage

1 Introduction

Recent trends in computer architecture and computer networks suggest that parallelism will pervade workstations, personal computers, and network clusters, causing parallelism to become available to more than just the users of traditional supercomputers. Experience with using parallelizing compilers and automatic parallelization tools has shown that these tools are often limited by the underlying sequential nature of the original program; explicit parallel programming by the user replacing sequential algorithms by parallel algorithms is often needed to take utmost advantage of these modern systems. A major obstacle to users in ensuring the correctness and reliability of their parallel software is the current lack of software testing tools for this paradigm of programming.

Researchers have studied issues regarding the analysis and testing of concurrent programs that use rendezvous communication. A known hurdle for applying traditional testing approaches to testing parallel programs is the nondeterministic nature of these programs. Some researchers have focused on solving this problem[13, 15], while others propose state-oriented program testing criteria for testing concurrent programs[14, 10]. Our hypothesis is that, with some extension, sequential test data adequacy criteria are still applicable to parallel program testing of various models of communication.

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.

Although many new parallel programming languages and libraries have been proposed to generate and manage multiple processes executing simultaneously on multiple processors, they can be categorized by their synchronization and communication mechanisms. Message passing parallel programming accomplishes communication and synchronization through explicit sending and receiving of messages between processes. Message passing operations can be blocking or nonblocking. Shared memory parallel programming uses shared variables for communication, and event synchronization operations.

In this paper, we focus on the applicability of one of the major testing criteria, all-du-path testing[16], to both shared memory and message passing parallel programming. In particular, we examine the problem of finding all-du-path coverage for testing a parallel program. The ultimate goal is to be able to generate test cases automatically for testing programs adequately according to the all-du-path criteria. Based on this criterion, all define-use associations in a program will be covered by at least one test case. The general procedure for finding a du-pair coverage begins with finding du-pairs in a program. For each du-pair, a path is then generated to cover the specific du-pair. Finally, test data for testing the path is produced [2][7]. This testing procedure has been well established for sequential programs; however, there is currently no known method for determining the all-du-path coverage for parallel programs. Moreover, the issues to be addressed toward developing such algorithms are not well defined.

We present our algorithms for shared memory parallel programs, and then discuss the modifications necessary for the message passing paradigm. We have been building a testing tool for parallel software, the **Delaware Parallel Software Testing Aid**, called **della pasta**, to illustrate the effectiveness and usefulness of our techniques. **della pasta** takes a shared memory parallel program as input, and interactively allows the user to visually examine the all-du-path test coverages, pose queries about the various test coverages, and modify the test coverage paths as desired. In our earlier paper, we focused strictly on the all-du-path finding algorithm[18].

We begin with a description of the graph representation of a parallel program used in our work. We then describe our testing paradigm and how we cope with the nondeterministic nature of parallel programs during the testing process. We discuss the major problems in providing all-du-path coverage for shared memory parallel programs, and a set of conditions to be used in judging the effectiveness of all-du-path testing algorithms. Current approaches to all-du-path coverage for sequential programs of closest relevance to our work are then discussed. We present our algorithm for finding an all-du-path coverage for shared memory parallel programs, which combines and extends previous methods for sequential programs. Modification of our data structures and algorithms for other parallel paradigms is discussed followed by the description of the **della pasta** tool. Finally, a summary of contributions and future directions are stated.

2 Program Model and Notation

The parallel program model that we use in this paper consists of multiple threads of control that can be executed simultaneously. A thread is an independent sequence of execution within a parallel program, (i.e., a subprocess of the parallel process, where a process is a program in execution). The communication between two threads is achieved through shared variables; the synchronization between two threads is achieved by calling *post* and *wait* system calls; and thread creation is achieved by calling the *pthread_create* system call.

We assume that the execution environment supports maximum parallelism. In other words, each thread is executed in parallel independently until a *wait* node is reached. This thread halts until a matching *post* is executed. The execution of *post* always succeeds without waiting for any other program statements.

Formally, a shared memory parallel program can be defined as follows: $\mathcal{PROG} = (T_1, T_2, \ldots, T_n)$, where $T_i, (1 \le i \le n)$ represents $n (\ge 2)$ threads. Moreover, T_1 is defined as the *manager* thread while all other threads are defined as *worker* threads, which are created when a *pthread_create()* system call is issued.

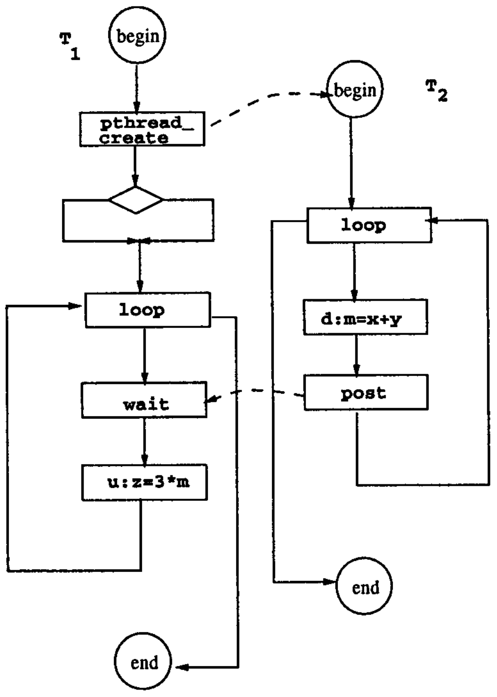


Figure 1: Example of a PPFG

To represent the control flow of a parallel program, a *Parallel Program Flow Graph* (PPFG) is defined to be a graph $G = (V,E)$ in which V 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 *pthread_create()* function, and n^j is the first statement in thread T_j $(T_i \neq T_j)$.

We define a *path* $P_i(n^i_{u_1}, n^i_{u_k})$ or simply 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).

Figure 1 illustrates a PPFG. All solid edges are intra-thread edges. Edges in E_S and E_T are represented by dotted edges. This diagram also shows a define node d of variable m, i.e., $m = x + y$, and a use node u, i.e., $z = 3 * m$. The sequence $begin - if - endif - loop - wait - u - loop - end$ is a path.

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 of a parallel program, we say that a node is *covered by a path*, denoted $n \in_p P$, if there exists a node n_s in the path such that $n = n_s$. We say that a node n^l $(1 \le l \le k)$ in a parallel 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 relation between the completion times of instances of two statement nodes. We say $a \prec b$ if an instance of the node a completes execution before an instance of the node b.

Finally, the problem of finding all-du-path coverage for testing a shared memory parallel program can be stated as: Given a shared memory parallel program, $\mathcal{PROG} = (T_1, T_2, \ldots, T_n)$, for each du-pair, (var, n^i_u, n^j_v), in $\mathcal{PROG}$, find a set of paths $PATH = (P_1, \ldots P_k)$ in threads $T_1, T_2, \ldots T_k$, that covers the du-pair (var, n^i_u, n^j_v), such that $n^i_u \prec n^j_v$.[1]

3 Nondeterminism and the Testing Process

Nondeterminism is demonstrated by running the same program with the same input and observing different behaviors, i.e., different sequences of statements being executed. Nondeterminism makes it difficult to reproduce a test run, or replay an execution for debugging. It also implies that a given test data set may not actually force the intended path to be covered during a particular testing run.

One way to deal with nondeterminism is to perform a controlled execution of the program, by having a separate execution control

[1]We focus on finding du-pairs with the define and use in different threads; du-pairs within the same thread are a subcase.

mechanism that ensures a given sequence of execution. We advocate controlled execution for reproducing a test when unexpected results are produced from a test, but we have not taken this approach to the problem of automatically generating and executing test cases to expose errors. Instead, we advocate *temporal testing* for this stage of testing.

We briefly describe our temporal testing paradigm here, and refer the reader to [19] for a more detailed description. *Temporal testing* alters the scheduled execution time of program segments in order to detect synchronization errors. Formally, a *program test case TC* is a 2-tuple ($\mathcal{PROG}$, I) where I is the input data to the program $\mathcal{PROG}$, whereas a *temporal test case TTC* is a 3-tuple ($\mathcal{PROG}$, I, $\mathcal{D}$) where the third component, referred to as *timing changes*, is a parameter for altering the execution time of program segments. Based on $\mathcal{D}$, the scheduled execution time of certain synchronization instructions n, represented as $\mathcal{D}(n)$, will be changed for each temporal test and the behavior of the program $\mathcal{PROG}$ will be observed.

Temporal testing is used in conjunction with path testing. For example, temporal all-du-path testing can be implemented by locating delay points along the du-paths being tested. The goal is to alter the scheduled execution time of all process creation and synchronization events along the du-paths. Delayed execution at these delay points is achieved by instrumenting the program with dummy computation statements. A testing tool is used to automatically generate and execute the temporal test cases. Similarly, new temporal testing criteria can be created by extending other structural testing criteria.

With the temporal testing approach, the testing process is viewed as occurring as follows:

(1) Generate all-du-paths statically.
(2) Execute the program multiple times without considering any possible timing changes.
(3) Examine the trace results. If the trace results indicate that different paths were in fact executed, it is a strong indication that a synchronization error has occurred and the du-path expected to be covered may provide some clue about the probable cause. Controlled execution may be used to reproduce the test. However, even if the same du-path was covered in multiple execution runs, temporal testing should still be performed.
(4) Generate temporal test cases with respect to the du-paths.
(5) Perform temporal testing automatically.
(6) Examine the results.

In this paper, we focus on the first step, i.e., developing an algorithm to find all-du-paths for shared memory parallel programs. The results of this paper can be used for generating temporal test cases with respect to the all-du-path coverage criterion. It should be noted that it is possible that the path we want to cover is not executed during a testing run due to nondeterminism, because we are not using controlled execution; instead, we use automatic multiple executions with different temporal testings to decrease the chances that the intended path will not be covered.

4 All-du-path Coverage

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

Figure 2 contains two threads, the `manager` thread and a `worker`

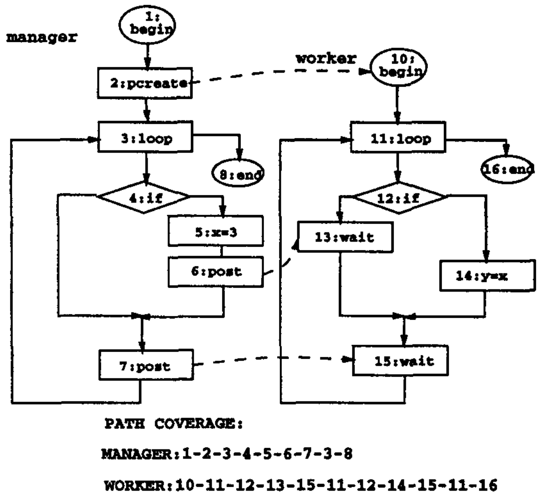


Figure 2: Du-pair coverage may cause an infinite wait.

thread. 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 a 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 a matching *post*, then the program will hang with the particular test case. In this example, the `worker` thread may not complete execution, whereas the `manager` thread will terminate successfully. The generated path will cause the loop in the `manager` thread to iterate only once, while the loop in the `worker` thread will iterate twice. This shows how the inconsistency in the number of loop iterations may cause one thread to wait infinitely. In addition, branch selection at an *if* node can also influence whether or not all threads will terminate successfully.

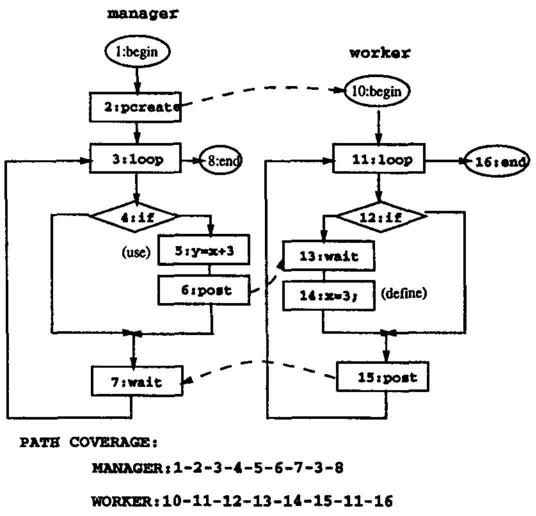


Figure 3: Du-pair is incorrectly covered.

In figure 3, the generated paths cover both the *define* (14) and the *use*

(5) nodes, 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 we should attempt to find a path coverage that will test this pair. The current path coverage does not accomplish this.

4.1 Test Coverage Classification

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

4.1.1 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 parallel program free of infeasible paths of the sequential programming kind (see the later section on 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 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, if only the *wait* is covered in a path coverage and a matching *post* is not, the path coverage is not a $PATH_a$. Figure 3, 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.

4.1.2 W-runnability of a du-path coverage

We have seen through the examples that a parallel program may cause infinite wait 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 that does not cause an infinite wait in any thread, we call the path coverage a *w-runnable du-path coverage*. When a *PATH* is w-runnable, we represent it as $PATH_w$. Although we call a *PATH* w-runnable, we are not claiming that a $PATH_w$ is free of errors, such as race conditions, or synchronization errors. More formally, a $PATH_a$ is w-runnable if all of the following additional conditions are satisfied:

1. 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.
2. $\nexists post$ nodes p^i, p^j, and *wait* nodes $w^i, w^j, \in_p PATH$ such that $((p^i \prec w^j) \vee (p^j \prec w^i)) \wedge (w^i \prec p^i) \wedge (w^j \prec p^j)$.

The first condition 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 *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 ensures that the generated path is free of deadlock.

We can develop algorithms to find $PATH_a$ automatically. However, we utilize user interaction in determining $PATH_w$ 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. We will find a $PATH_a$ and report that this path coverage may cause an infinite wait.

4.2 Infeasible Paths

Infeasible paths in a graph representation of a program are paths that will never be executed given any input data. In control flow graphs for a single thread, infeasible paths are due to data dependencies and conditionals. In interprocedural graph structures, infeasible paths are due to calling a function from multiple points. These kinds of infeasible paths can occur in sequential programs, and can also occur in parallel programs. In a parallel program, another kind of infeasible path can also occur due to synchronization dependencies. Infeasible paths due to synchronization dependencies can cause deadlock or infinite wait at run-time.

Like most path finding algorithms, we assume that the paths we identify are feasible with respect to the first causes. With regard to infeasible paths due to synchronizations, our work uses a slightly different characterization of paths, $PATH_a$ and $PATH_w$. Our du-path coverage algorithm finds paths in a way to guarantee that we will have matching synchronizations included in the final paths, that is, it finds paths that are $PATH_a$. However, a deadlock situation could occur for a path coverage that is a $PATH_a$, but not $PATH_w$. To guarantee finding matching synchronizations, we currently assume that matching *post* and *wait* operations both appear in a program. If a program contains a *post* and no matching *wait* or vice versa, we expect that the compiler will report a warning message prior to the execution of our algorithm.

5 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 [3, 11, 1]. All of these methods for finding actual paths focus on programs without parallel programming features and, therefore, cannot be applied directly to finding all-du-path coverage for parallel programs. However, we have found that the depth-first search approach and the approach of using dominator and post-dominator trees can be used together with extension to provide all-du-path coverage for parallel programs. We first look at their limitations for providing all-du-path coverage for parallel programs when used in isolation.

Gabow, Maheshwari, and Osterweil [3] 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, we claim that 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-du-path coverage for parallel programs.

Bertolino and Marrè have developed an algorithm (which we call DT-IT) that uses dominator trees (DT) and implied trees (IT) (i.e., post-dominator trees) to find a path coverage for all branches in a sequential program [1]. 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 we try to run this algorithm to find all-du-path coverage for parallel programs, we need to find a path coverage for all du-pairs instead of all-edges, 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 we are completing a path, we need to adjust the path just found to include the matching nodes. In addition, we will run into another problem regarding the order in which the *define* and *use* nodes are covered in the final path. For instance, in Figure 3, an incorrect path coverage will be generated using the DT-IT approach alone. The final path will have $define \not\prec use$. Thus, using this method alone cannot guarantee that we find a $PATH_a$.

Yang and Chung [20] 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.

6 A Hybrid Approach

In this section, we describe our extended "hybrid" approach to find the actual path coverage of a particular du-pair in a parallel program.

There are actually two disjoint sets of nodes in a path used to cover a du-pair in a parallel program: *required* nodes and *optional* nodes. The set of *required* nodes includes the *pthread_create()* calls as well as the *define* node and *use* node to be covered, and the associated *post* and *wait* with which the partial order $define \prec use$ is guaranteed. All other nodes on the path are *optional* nodes for which partial orders among them are not set by the requirements for a $PATH_a$. However, if a *wait* is covered by the path, a matching *post* must be covered. For instance, in figure 6, the nodes 2, 4, 7, 25, and 26 are required nodes, whereas all other synchronization nodes are optional. Among the required nodes, the partial orders are uniquely identified, whereas the partial orders among the optional nodes are not. For example, it is acceptable to include either $post_3$ or $post_4$ first in a path coverage. We can even include $wait_1$ later than $post_4$ in a $PATH_a$.

The DFS approach is most useful for finding a path that connects two nodes whose partial order is known. The DT-IT approach is most appropriate for covering nodes whose partial order is not known in advance. Therefore, DFS is most useful for finding a path between the *required* nodes, whereas the DT-IT approach is most useful for ensuring that the *optional* nodes are covered.

Our algorithm consists of two phases. During the first phase, called the *annotate phase*, the depth-first search (DFS) approach is employed to cover the required nodes in the PPFG. Then, the DT-IT approach is used to cover the optional nodes. After a path to cover a node is found, all nodes in the path are annotated with a *traversal control number* (TRN). In the second phase, called the *path generation* phase, the actual path coverage is generated using the traversal control annotations. We first describe the data structures utilized in the du-pair path finding algorithm, and then present the details of the algorithm.

The algorithm assumes that the individual du-pairs of the parallel program have been found. Previous work computing reaching definitions for shared memory parallel programs has been done by Grunwald and Srinivasan [5].

6.1 Data Structures

The main data structures used in the hybrid algorithm are: (1) a PPFG, (2) a working queue per thread to store the *post* nodes that are required in the final path coverage, (3)a traversal control number(TRN) associated with every node used to decide which node must be included in the final path coverage and how many iterations are required for a path through a loop, (4) a reverse post-order number (RPO) for each node in the PPFG used in selecting a path at loop nodes, (5) a decision queue per *if-node*, and (6) one path queue per thread to store the resulting path.

6.2 The Du-path Finding Algorithm

We describe the du-path finding algorithm with respect to finding du-pairs in which the define and use are located in different threads. The handling of du-pairs with the define and use in the same thread is a simplification of this algorithm. Figure 4 contains the *annotate_the_graph()* algorithm, which accomplishes the annotate phase. The *traverse_the_graph()* algorithm, shown in Figure 5, traverses the PPFG and generates the final du-path coverage. We describe each step of these algorithms in more detail here.

Phase 1: Annotating the PPFG.
Step 1. Initialize the working and decision queues to empty, and set TRN of each node to zero.
Step 2. Use DFS to find a path from the *pthread_create* of the thread containing the *define* node to the *define* node, and then from the

```
Algorithm annotate_the_graph()
Input: A DU-pair, and a PPFG
Output: Annotated PPFG
Method:
  1. Initialize TRN's, decision queues, and working queues;
  2. Find a path to cover pthread_create and define nodes using dfs;
     From the define node, search for the use node using dfs;
  3. Complete the two sub-paths using DT-IT.
  4. For each node in the complete paths:
     Increment TRN by one;
     If node is a WAIT,
       Add matching nodes into appropriate working queues,
     If node is an if-node,
       Add the successor node in the path into decision queue;
  5. /* process the synchronization nodes */
     while ( any working queue not empty )
     {  For each thread, if working queue not empty
        {  Remove one node from the working queue;
           if the node's TRN is zero
           {
              Find a path to cover this node
              For each node in the complete path:
                Increment TRN by one;
                If node is a WAIT,
                  Add matching nodes into appropriate working queues,
                If node is an if-node,
                  Add the successor node in the path into decision queue;
           }
        }
     }
```

Figure 4: Phase 1: Annotate the graph.

define node to the *use* node. When a *post* node is found in the path, a matching *wait* is placed as the next node to be traversed, and the search for the *use* node continues. Upon returning from each DFS() call after a *wait* is traversed, return to the matching *post* before continuing the search for the *use* node if not yet found.

Step 3. Apply DT-IT to complete the sub-paths found in Step 2. To complete the sub-path in the thread containing the *define* node, use the dominator tree of the *define* node and the post-dominator tree of the *post* node that occurs after the *define* node in the sub-path just found. Similarly, to complete the sub-path in the thread containing the *use* node, use the dominator tree of a matching *wait* of the *post* node and the post-dominator tree of the *use* node.

Step 4. For each node covered by either of these two paths,

(1) increment the node's TRN by one to indicate that the node should be traversed at least once.

(2) If the node is a *wait*, add a matching *post* into the working queue of the thread where the *post* is located.

(3) If the node is an *if-node*, add the Reverse Post-Order Number(RPO) of the successor node within the path into the *if-node*'s decision queue to ensure the correct branch selection in phase 2.

Step 5. While any working queue is not empty, remove one *post* node from a thread's working queue, and find a path to cover the node. Increment the TRN of the nodes in that path. In this way, the TRN identifies the instances of each node to be covered. This is particularly important in finding a path coverage for nodes inside loops, where it might be necessary to traverse some loop body nodes several times to ensure that branches inside the loop are covered appropriately. Process *wait* and *if*-nodes in this path as in Step 4.

```
Algorithm traverse_the_graph()
Input: An annotated PPFG
Output: A DU-path
Method:
For all threads
{
   current = begin node of the thread;
   while ( current node's TRN > 0 and
           current is not the end node )
   {
      add the current node to the result DU-path;
      decrement TRN of current node by one;
       if ( current is an if-node )
          current = first node from decision queue;
          delete the first node in the queue;
      else
       if ( current is a loop node )
          current = successor with smallest non-zero RPO;
       else
            current = successor node of current;
   }
}
```

Figure 5: Phase 2: Generate the du-path coverage

Phase 2: Generating a du-path. For each thread, perform the following steps:

Step 1. Let n be the *begin* node of the thread.

Step 2. While n's TRN > 0 and n is not the *end* node,

Add n to the path queue, which contains the resulting path coverage, and decrement n's TRN. If n is an *if-node*, then let the new n be the node removed from n's decision queue. Otherwise, if n is a *loop* node, the successor with the smallest non-zero TRN is chosen to be the new n. If the children have the same TRN, then the child with the smallest RPO is chosen. Otherwise, if n is not an *if-node* or *loop* node, let new n be the successor of n.

6.3 Examples

In this section, we use two examples to illustrate the hybrid approach. The first example illustrates generating a $PATH_w$, while the second example illustrates generating a (non-$PATH_w$) $PATH_a$. Both examples cover the du-pair with the define of X at node 4 and the use of X at node 26 in Figure 6.

Example 1 Generating a $PATH_w$:

During the second step of the first phase, the required nodes, including the *pthread_create*, *define*, $post_2$, $wait_2$, and the *use* nodes, are included in a partial path. The identified partial path is 2-3-4-5-7-25-26. During the third step of the first phase, the two sub-paths are completed, using the DT-IT approach. The two identified complete paths are 1-2-3-4-5-7-8-9-3-11 for *manager* and 21-22-23-25-26-27-28-22-30 for $worker_1$. The TRN for every node along the two paths equals 1 after step 4 except the loop node 22 for which the TRN is 2. When node 9 was reached during this traversal, nodes 28 and 35 were put into the working queues for $worker_1$ and $worker_2$, respectively. When node 28 is taken out of the working queue in step 5, it is found to have a nonzero TRN, and thus no more paths are added. When node 35 is taken out of the working queue, the TRN is zero. Hence, the path 31-32-33-34-35-32-36 is found to cover node 35. With the annotated PPFG as input, the second phase finds a final path of 1-2-3-4-5-7-8-9-3-11 for *manager*, 21-22-23-25-26-27-28-22-30 for $worker_1$, and 31-32-33-34-35-32-36 for $worker_2$.

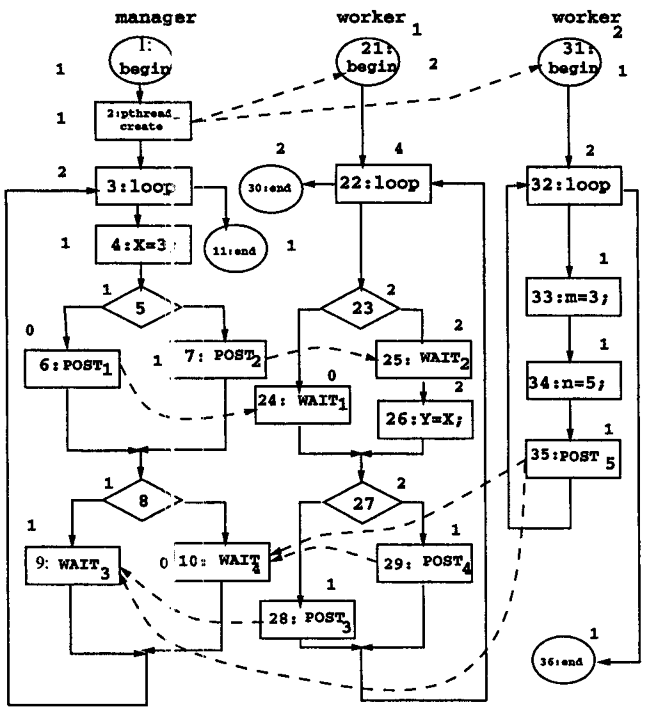


Figure 6: Example of the path finding algorithm

Example 2 Generating a non-$PATH_w$:
During the second step of the first phase, the required nodes, including the *pthread_create*, *define*, *post*$_2$, *wait*$_2$, and the *use* nodes, are included in a partial path identified as 2-3-4-5-7-25-26. During the third step of the first phase, the two sub-paths are completed, and found to be 1-2-3-4-5-7-8-9-3-11 for *manager* and 21-22-23-25-26-27-29-22-30 for *worker*$_1$. The TRN for every node along the two paths equals 1 after step 4 except for the loop node 22; the TRN for node 22 equals 2. When node 9 was reached during this traversal, nodes 28 and 35 were put into the working queues. Similarly, during the fifth step of the first phase, two paths 21-22-23-25-26-27-28-22-30 and 31-32-33-34-35-32-36, are found to cover nodes 28 and 35, respectively. The final TRN's for this example label each node in Figure 6. The second phase finds final paths of 1-2-3-4-5-7-8-9-3-11 for *manager*, 21-22-23-25-26-27-29-22-23-25-26-27-28-22-30 for *worker*$_1$ and 31-32-33-34-35-32-36 for *worker*$_2$. This set of paths is not *w-runnable* because *worker*$_1$ has an infinite wait (at node 25).

It should be noted that regardless of the path constructed, the user will have to validate that the *w* property holds.

6.4 Correctness and Complexity

Given a du-pair in a parallel program where the *define* node and the *use* node are located in two different threads, we show that this algorithm indeed will terminate and find a $PATH_a$. We first introduce some lemmas before we give the final proof.

Lemma 1: *TRN preserves the number of required traversals of each node within a loop body.*

During the first phase, the TRN of a node is incremented by one each time a path is generated that includes that node. Therefore, the number of traversals of each node in paths found during the first phase is preserved by the TRN. Although the number of traversals during the first phase is preserved, we are not claiming that these nodes will indeed be traversed during the second phase that same number of times. For nodes outside of a loop body, each node will be traversed at most as many times as its TRN. But a node may not need to be traversed that many times because the path generation phase may reach the *End* node before the TRN of all nodes becomes zero. Moreover, if there is no loop node in a program, only required nodes will be traversed as many times as the TRN indicates.

Lemma 2: *The decision queue and TRN of an* if-node *guarantee that the same sequence of branches selected during the first phase will be selected during the second phase.*

When an *if-node* is found in a path during the first phase, one branch is stored into the decision queue at that time. Hence, the number of branches in the decision queue of a given *if-node* is equal to the TRN of that *if-node*. Each time the *if-node* is traversed during the second phase, one node is taken out of the decision queue and the TRN of the *if-node* is decremented by one. Therefore, the sequence that a branch is selected is preserved.

Lemma 3: *DFS used during the first phase ensures define $\prec$ post $\prec$ wait $\prec$ use in the final generated path.*

During the first phase, the required nodes will be marked by DFS prior to any other nodes in the graph. This ensures that necessary branches are stored in the decision queues first. By Lemma 2, these branches will be traversed first during the second phase. Hence, these nodes will be traversed in the correct order as given by the relationships above. Therefore, Lemma 3 is valid.

Lemma 4: *The working queues and TRN together guarantee the termination of the Du-path Finding Algorithm.*

We must show that both phases terminate.
Phase 1 Termination: We use mathematical induction on m, where m represents the total number of pairs of synchronization nodes covered in a path coverage.
Base case: $m = 1$. Since there is only one pair of synchronization calls, the required ones, they will be included in the path generated by the DFS. The completion of the two partial paths will automatically terminate since there are no extra *post* or *wait*'s involved. Now, assume Lemma 4 is true when $m = k$ where k is an integer greater than 1. We need to show that Lemma 4 is also true when $m = k+1$. If the *post* and *wait* have been traversed previously, the TRN of these nodes will be greater than zero. Hence, they will not be included again during the first phase. When we generate a new path to cover this pair of *post* and *wait* nodes, if they currently have TRN=0, all other pairs of synchronization nodes will have been covered. (by the induction step) Hence, this new pair of synchronization calls will not trigger an unlimited number of actions. Therefore, the annotation phase will terminate.
Phase 2 Termination: Since the TRN for each node must be traversed is a finite integer, and the TRN is decremented each time it is traversed during phase 2, the traversal during phase 2 will not iterate forever. Whenever a node with zero TRN or the *End* node is reached, the path generation phase terminates.

Finally, we show the proof of the following theorem.

Theorem 1: *Given a du-pair in a shared memory, parallel program, the hybrid approach terminates and finds a $PATH_a$.*

Proof: (1) By Lemma 4, the hybrid approach terminates. (2) To show that a $PATH_a$ is generated, we must show that the conditions described in the definition of $PATH_a$ are satisfied. By Lemma 1, Lemma 2, and Lemma 3, we can conclude that the *define*, *use*, the required *post*, and *wait* nodes will be covered in the correct order. Step 1 of Phase 1 ensures that all appropriate *pthread_create* calls are covered. Step 5 ensures that a matching *post* node regarding each *wait* node included in the path is also covered. Therefore, all conditions for a $PATH_a$ are satisfied. Q.E.D.

The running time of the hybrid approach includes the time spent searching for the required nodes and time spent generating the final path coverage. We assume that the dominator/implied trees and the du-pairs have been provided by an optimizing compiler.

Theorem 2: For a given $G = (V,E)$, and a du-pair (d, u), the total running time of the du-path finding algorithm is equal to $O(2*k*(|V|+|E|))$, where the total number of *post* or the *wait* calls is denoted by k.

Proof: The running time for searching for the required nodes is equal to $O(|V|+|E|)$. To complete the two partial paths, the running time is equal to $O(2*k*(|V|+|E|))$, where the total number of *post* or the *wait* calls is denoted by k. Finally the second phase takes time $O(2*k*(|V|+|E|))$ to finish. Hence, the total running time is equal to $O(2*k*(|V|+|E|))$. For a given graph, usually the number of edges is greater than that of the nodes. Then, the running time is equal to $O(2*k*|E|)$. Q.E.D.

7 Other Parallel Paradigms

7.1 Rendezvous communication

Among other researchers, Long and Clarke developed a data flow analysis technique for concurrent programs[9]. After their data flow analysis is performed, we can apply a modified version of our algorithm to find all-du-path coverage for a concurrent program with rendezvous communication. In particular, we need to modify the following: (1) construction of the PPFG, (2) definition of path acceptability, and (3) the all-du-path finding algorithm.

First, to accommodate the *request* and *accept* operations in concurrent programs to achieve rendezvous communication, the PPFG needs to include a directed edge from a *request* to an *accept* node. Secondly, since the execution of a *request* is synchronous, the second condition in the definition of $PATH_a$ must be replaced by the following two conditions:

2a. $\forall$*accept* nodes $a \in_p PATH$, $\exists$ a *request* node $r \in MR(a)$, such that $p \in_p PATH$, and

2b. $\forall$*request* nodes $r \in_p PATH$, $\exists$ a *accept* node $a \in MA(r)$, such that $a \in_p PATH$.

The set $MR(a)$ is defined as the set of *matching requests for the accept node a*; the set $MA(r)$ is defined as the set of *matching accepts for the request node r*.

Finally, during the first phase of the algorithm, whenever a *request* or an *accept* is found, the matching node must be added into the working queue.

7.2 Message Passing Programs

For analyzing message passing programs, a data flow analysis similar to interprocedural analysis for sequential programs is needed to compute the define-use pairs across processes. Several researchers have developed interprocedural reaching definitions data flow analysis techniques, even in the presence of aliasing in *C* programs [12, 6]. Although this analysis may find define-use pairs that may not actually occur during each execution of the program, the reaching definition information is sufficient for program testing. After this information is computed, we can apply our algorithm to find all-du-path coverage for a given *C* program with message passing library calls. The Message Passing Interface(MPI)[4] standard is a library of routines to achieve various types of inter-process communication, i.e., synchronous or asynchronous *send/receive* operations.

To find all-du-path coverage for message passing programs, we need to identify the type of *send* or *receive* operations first, i.e., synchronous or asynchronous. If the *send* operation is synchronous, the definition of a $PATH_a$ must be modified to include both the *send* and the matching *receive* in the path coverage similar to the change made for supporting rendezvous-communication parallel programs. If the *send* is asynchronous, we only need to replace *post* by *send* in this paper. For each synchronous *receive* operation, we need to replace the *wait* by a *receive* in our algorithms and definitions.

8 The *della pasta* Tool

The algorithm described in this paper has been incorporated into **della pasta**, the prototype tool that we are building for parallel software testing. The objective is to demonstrate that the process of test data generation can be partially automated, and that the same tool can provide valuable information in response to programmer queries regarding testing. The current major functions of this tool are: (1) finding all du-pairs in the parallel program, (2) finding all-du-path coverage to cover du-pairs specified by the user, (3) displaying all-du-path coverage in the graphic or text mode as specified by the user, and (4) adjusting a path coverage when desired by the user.

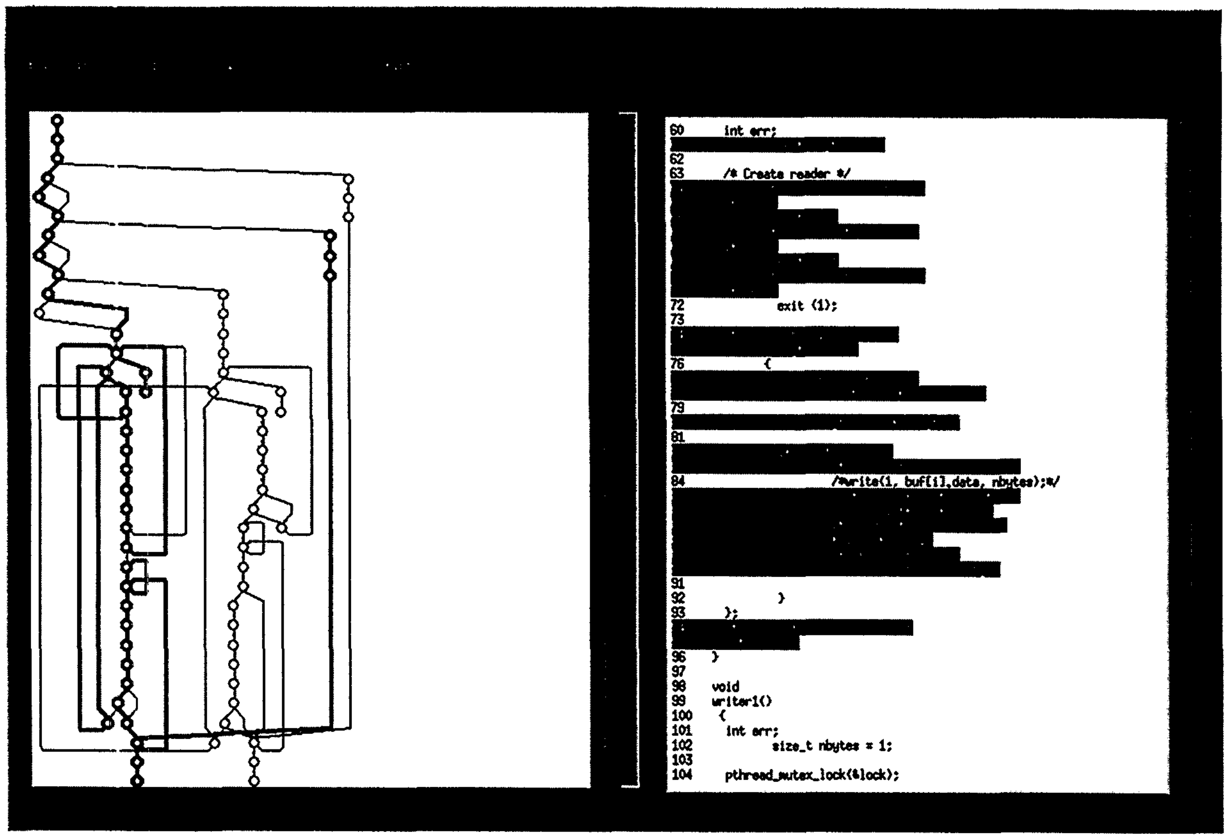
della pasta consists of two major components: the *static analyzer* which accepts a file name and finds all du-pairs as well as the all-du-path coverage for each du-pair, and the *path handler* which interacts with the user to display the PPFG, a path coverage, and accept commands for displaying individual du-pair coverages and for modifying a path. The static analyzer uses a modified version of the Grunwald and Srinivasan algorithm[5] to find du-pairs in parallel programs of this model, and is implemented using the compiler optimizer generating tool called *nsharlit*, which is part of the SUIF compiler infrastructure [8]. The path handler is built on top of *dflo* which is a data-flow equation visualizing tool developed at Oregon Graduate Institute. [2]

The user interface of **della pasta** is illustrated in figure 7. On the left of the screen, the PPFG is illustrated; on the right, the corresponding textual source code is shown. A user can resize the data flow graph as desired. The currently selected def-use pair is shown at the top of the screen. The corresponding du-pair path coverage is depicted in the PPFG as well as in the text as highlighted nodes and statements, respectively. Clicking on any node in the PPFG will pop up an extra window with some information about the node, and allow the user to modify a path coverage.

In this example, a reader/writer program is illustrated in which the main thread creates three additional threads: two readers and one writer. The main thread then acts as one writer itself and communicates with one of the two readers just created. These two pairs of readers/writers will work independently in parallel. The du-pair coverage shown in this example only involves two of the 4 threads in the program.

We are currently extending **della pasta** to use the du-pair coverage

[2]This tool can be downloaded from the Internet. Refer to the web site http://www.cse.ogi.edu:80/Sparse/dflo.html for details.



```
    int err;
[illegible]

    /* Create reader */
[illegible]
            exit (1);

[illegible]
          {
[illegible]

[illegible]

[illegible]
                       /*write(1, buf[i].data, nbytes);*/
[illegible]

          }
     };
[illegible]
 }

 void
 writer1()
  {
   int err;
          size_t nbytes = 1;

   pthread_mutex_lock(&lock);
```

Figure 7: della pasta user interface

information already available through our static analyzer to answer queries of the following kind: Will the test case execute successfully without infinite wait caused by the path coverage? What other du-pairs does a particular path coverage cover? We are also incorporating our temporal testing techniques [17] into the tool in order to provide testing aid for delayed execution in addition to the traditional all-du-path testing.

9 Summary and Future Work

To our knowledge, this is the first effort to apply a sequential testing criterion to shared memory or message passing parallel programs. Our contributions include sorting out the problems of providing all-du-path coverage for parallel programs, classifying coverages, identifying the limitations of current path coverage techniques in the realm of parallel programs, developing an algorithm that successfully finds all-du-path coverage for shared memory parallel programs, showing that it can be modified for message passing and rendezvous communication, and demonstrating its effectiveness through implementation of a testing tool.

The all-du-path coverage algorithm presented in this paper has some limitations. The all-du-path algorithm requires that a PPFG be constructed statically. If a PPFG cannot be constructed statically to represent the execution model of a program, the analysis that constructs the du-pairs may not produce meaningful du-pairs. Thus, the number of `worker` threads is currently assumed to be known at static analysis time. In the case where a *clear* operation is used to clear an event before or after the *wait* is issued, our analysis will report more du-pairs than needed. In testing, this only implies that we indicate more test cases than really needed.

We are in the process of examining these limitations, while experimentally analyzing the effectiveness of fault detection for parallel programs using the all-du-paths criterion with **della pasta**, and investigating other structural testing criteria for testing parallel programs.

Acknowledgements

We would like to thank Barbara Ryder for her helpful comments in preparing the final paper.

"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] A. Bertolino and M. Marrè. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Trans. on Soft. Eng.*, 20(12):885–899, Dec. 1994.

[2] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. on Soft. Eng.*, 2(3):215–222, Sept. 1976.

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

[4] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1994.

[5] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 159–168, California, USA, 1993.

[6] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, Mar. 1994.

[7] B. Korel. Automated software test data generation. *IEEE Trans. on Soft. Eng.*, 16(8):870–879, Aug. 1980.

[8] M. S. Lam. Introduction to the SUIF compiler system. In *First SUIF Compiler Workshop*, Jan. 1996.

[9] 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, Dept of Computer Science, July 1991.

[10] S. Morasca and M. Pezzè. 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.

[11] S. C. Ntafos and S. L. Hakimi. On path cover problems in digraphs and applications to program testing. *IEEE Trans. on Soft. Eng.*, 5(5):520–529, Sept. 1979.

[12] H. D. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Trans. on Soft. Eng.*, 20(5), 1994.

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

[14] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Trans. on Soft. Eng.*, 18(3):206–215, Mar. 1992.

[15] 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.

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

[17] C. Yang and L. L. Pollock. The challenges in automated testing of multithreaded programs. In *the 14th International Conference on Testing Computer Software*, pages 157–166, June 1997.

[18] 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*, Nov. 1997.

[19] C.-S. D. Yang and L. L. Pollock. Semi-automatic temporal testing for parallel programs. Technical Report 98-05, U. of Delaware, Dept of CIS, Sept. 1997.

[20] R.-D. Yang and C.-G. Chung. Path analysis testing of concurrent programs. *Information and Software Technology*, 34(1):43–56, Jan. 1992.