An Algorithm for All-du-path Testing Coverage of Shared Memory Parallel Programs

C.-S. D. Yang and L. L. Pollock
yang@cis.udel.edu pollock@cis.udel.edu
Computer and Information Sciences
University of Delaware
Newark, DE 19716, USA

Abstract

Little attention has focused on applying traditional testing methodology to parallel programs. This paper discusses issues involved in providing all-du-path coverage in shared memory parallel programs, and describes an algorithm for finding a set of paths covering all define-use pairs. To our knowledge, this is the first effort of this kind.

1 Introduction

As computer systems rapidly evolve towards higher performance through the use of parallelism, new programming languages and libraries are providing users with the capabilities to generate and manage multiple processes executing simultaneously on multiple processors. In addition to sequential computation, parallel programs explicitly or implicitly involve process creation and communication and synchronization between processes. As these new paradigms gain popularity, the demand for, and the corresponding lack of, program testing tools for parallel programming becomes more evident. This paper describes contributions towards the goal of providing automatic generation of test cases for the structural testing of parallel programs.

Structural testing of sequential programs has been studied extensively [4, 5, 8, 1]. However, there is currently no known method for determining the all-du-path coverage for parallel programs. While the general procedure for finding all-du-paths for shared memory parallel programs can be similar to that for sequential programs, the support of more general synchronization language primitives, e.g., *post* and *wait*, makes the task of finding all-du-path coverage more complicated.

[0]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.

In this paper, we describe a method to find all-du-path coverage for testing shared memory parallel programs. We begin by defining the problem in section 2, with a description of the graph representation of a parallel program, illustration of the major issues, and a set of conditions to be used in judging the effectiveness of all-du-path testing algorithms. The limitations of using existing methods for generating all-du-path coverage for sequential programs are then investigated in section 3, along with the capabilities of these methods that might be useful in some aspect of providing all-du-path coverage for parallel programs. In section 4, a "hybrid" algorithm for finding an all-du-path coverage based on dominator and implied trees(DT-IT), depth-first search(DFS), and a traversal control number is presented, exemplified, and analyzed. This paper is concluded by a summary of the contributions and future work in section 5.

2 Defining the Problem

A parallel program consists of multiple threads of control which 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 model of parallel programming that we target in this work is shared memory parallel programming in which communication is achieved through shared variables, and the synchronization between two threads is achieved by calling *post* and *wait* system calls; the thread creation is achieved by calling the *pthread_create* system call. We assume that the execution environment supports the maximum parallelism. In other words, each thread is executed in parallel independently until a *wait* node is reached. This thread halts until the 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: $\mathcal{P}ROG = (T_1, T_2, \ldots, T_n)$, 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.

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 $i \ne j$. 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 $(T_i \ne T_j)$.

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

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

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

We represent the set of matching posts of a wait node as $MP(w) = \{p|(p, w) \in E_S\}$ and the set of matching waits of a post node as $MW(p) = \{w|(p, w) \in E_S\}$. We use the symbol "$\prec$" to represent the Lamport's *happen before* relation between two nodes[6]. We say $a \prec b$ if the node a is executed before 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, $\mathcal{P}ROG = (T_1, T_2, \ldots, T_n)$, for each du-pair, (var, n^i_u, n^j_v), in $\mathcal{P}ROG$, 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 $n^i_u \prec n^j_v$.

2.1 Issues

In this section, issues regarding all-du-path coverage are demonstrated. This list is not necessarily exhaustive, but instead meant to illustrate the complexity of the problem.

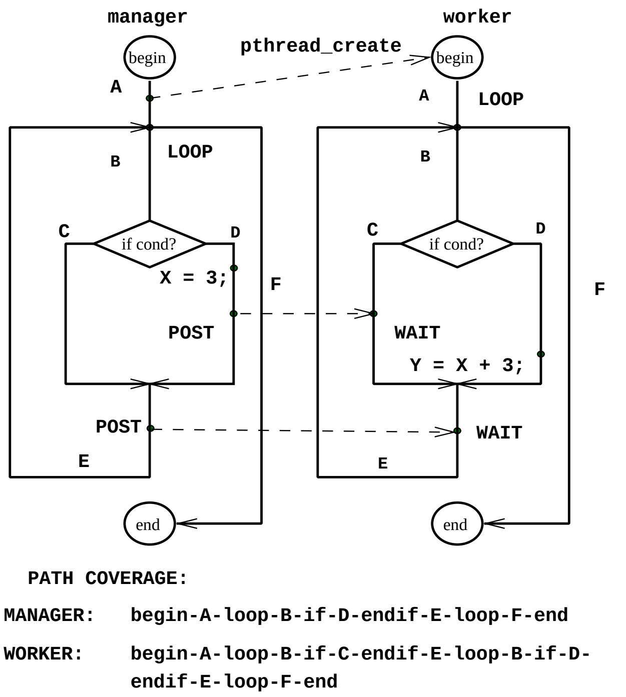


Figure 1: Du-pair coverage may cause an infinite wait.

In Fig.1, the generated paths cover the du-pair correctly because the *define* node will be executed prior to the *use* node. However, the `worker` thread may not complete execution. 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. Similarly, at some *if* nodes, the selection of a correct branch may also be crucial to the execution results.

If a *post* is covered and not the matching *wait*, the program will execute to completion, despite the fact that the synchronization is not covered completely. However, if the *wait* is covered and not the matching *post*, then the program will hang with the particular test case.

In Fig.2, the generated paths cover both the *define* and the *use* nodes, but the *use* node will be reached before the *define* node, that is, $use \prec define$.

Finding du-pair coverages for situations like these where there are loops, *if* nodes, and matching *post*'s and *wait*'s inside the loops, is not straightforward due

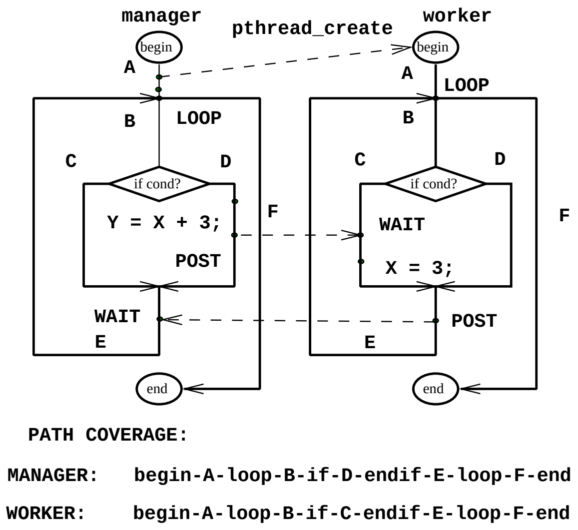


Figure 2: Du-pair is incorrectly covered.

to the problems seen in covering the matching *post* and *wait* involved in the paths.

2.2 Requirements

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.*

2.2.1 Acceptability of a du-path coverage

We call a set of paths $PATH$ an *acceptable du-path coverage*, denoted as $PATH_a$, for the du-pair ($define$, *use*) in a 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 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 new thread, the associated thread creation node must also be included in the path coverage. If any of these conditions is violated, then the path coverage is considered to be *unacceptable.*

2.2.2 W-runnability of a du-path coverage

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.* denoted as $PATH_w$. 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. $\not\exists post$ 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 more difficult cases. The reason is that in some cases the $PATH_w$ may not exist Hence, finding the $PATH_w$ is not computationally possible.

3 Current Approaches

In the context of sequential programs, several researchers have contributed to generating test cases using path finding as well as finding minimum path coverage [2, 7, 1]. 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 DFS and the DT-IT approaches 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.

3.1 Depth-first Search (DFS)

Gabow, Maheshwari, and Osterweil [2] propose to use DFS to find actual paths for connecting two nodes in sequential programs. When applying the 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 the 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 the 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.

3.2 Dominator/Implied Trees (DT-IT)

Bertolino and Marrè [1] have developed an algorithm that uses dominator trees (DT) and implied trees (IT) (i.e., post-dominator trees) to find a path coverage for all branches in sequential programs [1]. If we apply this algorithm alone to find the 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 change. However, we also run into the same problem as in the 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 have a problem regarding the order in which the *define* and *use* nodes are covered in the final path. For instance, in Figure 2, an incorrect path coverage will be generated using the DT-IT approach alone. The final path will have $use \prec define$. Thus, using this method alone cannot even guarantee that we find a $PATH_a$.

4 A Hybrid Approach

In this section, we describe an extended "hybrid" approach to find the actual path coverage of a particular du-pair in a parallel program. The algorithm consists of two phases. During the first phase, the DFS approach is employed to cover the required nodes, the *define*, *use*, and the *post* and *wait* that associate with the *define* and *use* nodes, in the PPFG. Then, the DT-IT approach is used to cover other *post* and *wait* nodes as needed. After a path is found for covering a node, all nodes in the path are annotated with a *traversal control number* (TRN). In the second 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. In our implementation, we use the work by Grunwald and Srinivasan [3] as the basis for finding individual du-pairs.

4.1 Data Structures

The main data structures used in the hybrid algorithm are: (1) a PPFG and a source level control flow graph, (2) one *post/wait* working queue per thread used for storing the *post/wait* nodes that are required to be included in the final path coverage, (3) a traversal control number(TRN) associated with every node and 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) one decision queue per *if-node*, and (6) one path queue per thread used for storing the resulting path.

4.2 The Du-path Finding Algorithm

The algorithm consists of two phases: an *annotate phase* and a *path generation phase*. Figure 3 contains the function *annotate_the_graph()*, which accomplishes the annotate phase. This phase calls the function *process_the_syn_nodes()*, shown in Fig.4, to search each path in the set of annotated paths to find matching synchronization pairs as required for $PATH_a$. The function *traverse_the_graph()*, shown in Fig.5, traverses the PPFG and generates the final du-path.

4.3 An Example

The text below Fig.6 shows the results of the phases of the du-finding algorithm for two different paths that could be generated by the algorithm to cover the same du-pair in the figure. A $PATH_w$ and a non-$PATH_w$ generated by the algorithm are examined. For each path example, the text gives the paths generated after phase 1 and 2, the TRN's of the nodes, and finally the final $PATH_a$ generated by phase 2.

4.4 Analysis of the Algorithm

In this section, we prove that this algorithm terminates and finds a $PATH_a$. We introduce some lemmas (without proofs due to space constraints) before we present the final theorem.

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

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

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

Algorithm annotate_the_graph()
Input: A DU-pair, a PPFG and a control flow graph
Output: Annotated control flow graph
Method:

Initialize TRN's, Decision Queues, and Working Queues;

Find the pthread_create node in all threads;

Find a path to cover the pthread_create and the define node,
then from the define node search for the use node using dfs();
If a post is found and the use is found in the reach_OUT of
the post, prune all branches, then resume the search at the
matching wait until the use node is found.

Complete the two sub-paths using DT-IT approach.
Increment the TRN of each node in the path by one;

For each POST/WAIT node in the path set,
Add matching nodes into the appropriate thread's working queue;
For each if-node in the path set,
Add the successor nodes into the decision queue;

Call process_the_syn_nodes;

Figure 3: Phase 1: annotate the graph.

Algorithm process_the_syn_nodes()
Input: A path, and a PPFG
Output: Annotated control flow graph
Method:

```
while ( any working queue not empty )
{
  loop through all threads
  {
    Remove one node from the working queue;
    if the node's TRN is zero
    {
        Find a path to cover this node
            and increment the TRN of each node
            in the path by one;
        For each POST/WAIT in the path
            Add matching nodes into
            appropriate thread's working queue;
        if this node is a decision node, i.e., an if-node,
            add the successor node to the decision queue;
        Call process_the_syn_nodes;
    }
  }
}
```

Figure 4: Process synchronization nodes

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

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 DT, IT, 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.

5 Summary and Future Work

To our knowledge, this is the first algorithm for finding all-du-path coverage in shared memory parallel programs. The major contributions of this research are in three directions. First, we identify the issues in providing all-du-path coverage for parallel programs as well as the problems in using the DFS or DT-IT approaches individually to find all-du-paths for shared memory parallel programs. Second, we propose a hybrid algorithm which can be applied to find $PATH_a$. Finally, this algorithm can provide some valuable information for answering questions such as: What kind of du-path coverage is provided? Will it be executed successfully? Is it possible to cause an infinite wait? What other du-pairs does a particular path coverage cover? Currently, we are in the process of incorporating this algorithm into a tool that can assist a tester

Algorithm traverse_the_graph()

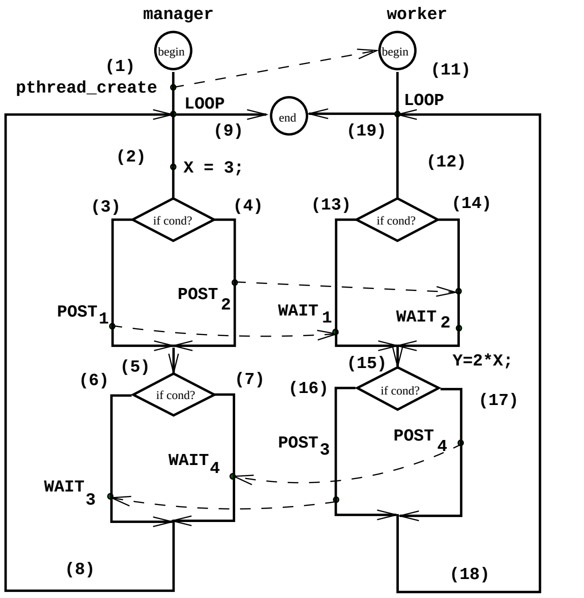
Input: An annotated control flow graph and the PPFG

Output: A DU-path

Method:

```
For all threads
{
    current = source node of the thread;
    while ( current node's TRN > 0 and
            current is not the sink node )
    {
        add the current node to the result DU-path;
        decrement the TRN of the current node by one;
        if ( current is a decision node )
            current = pop one node out of the decision queue;
        else
        if ( current is a loop node )
            current = successor with the smallest non-zero RPO;
        else
            current = the successor node of current;
    }
}
```

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



Generating $PATH_W$

1.(after phase 1)
manager:1-2-4-5-7-8-9
worker:
11-12-14-15-17-18-19
TRN of nodes along the paths=1

2.(after phase 2)
manager:1-2-4-5-7-8-9
worker:
11-12-14-15-17-18-19

Generating non-$PATH_W$

1.(after phase 1)
manager:1-2-4-5-6-8-9
worker(1st iteration):
11-12-14-15-17-18-19
worker(2nd iteration):
11-12-14-15-16-18-19
TRN of 11,12,14,15,18,19=2
TRN of nodes 16,17=1

2.(after phase 2)
manager:1-2-4-5-6-8-9
worker:
11-12-14-15-17-18-12-14-15-16-18-19

Figure 6: Example of the Hybrid Approach

in automating the process of test suite generation and assess the percentage of coverage after each testing is performed.

We are also investigating testing for other models of parallel programming. So far, we have concentrated on testing in a restricted shared memory parallel programming model that includes the programming features *post*, *wait*, and *pthread_create*. There are other parallel programming features that we currently ignore. For example, the *barrier* and *forall* are not supported by our algorithm yet. We also assume that the parallel program is well-structured using *if-statements*, *while-loops*, and *assignment statements*, not unstructured goto's. Moreover, the number of `worker` threads is currently assumed to be known at static analysis time. We also assume that there are no *infeasible paths*.

Lastly, we intend to analyze the effectiveness of fault detection for parallel programs using the all-du-paths criterion, and investigate other structural testing criteria for testing parallel programs.

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed 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 Transactions on Software Engineering*, 20(12):885–899, Dec. 1994.

[2] 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, Sept. 1976.

[3] 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.

[4] J. R. Horgan and S. London. Data flow coverage and the C language. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 87–97, Oct. 1991.

[5] R. Jasper, M. Brennan, K. Williamson, and B. Currier. Test data generation and feasible path analysis. In *Proceedings*

of the International Symposium on Software Testing and Analysis, pages 95–107, Seattle, Washington, Aug. 1994.

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

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

[8] P. T. Revanbu, D. S. Rosenblum, and A. L. Wolf. Automated construction of testing and analysis tools. In *International Conference on Software Engineering*, pages 241–250, Sorrento, Italy, 1994.