Structural Testing of Concurrent Programs

Richard N. Taylor, David L. Levine, and Cheryl D. Kelly

Abstract—Although structural testing techniques are among the weakest available with regard to developing confidence in sequential programs, they are not without merit. This paper extends the notion of structural testing criteria to concurrent programs and proposes a hierarchy of supporting structural testing techniques. Coverage criteria described include concurrency state coverage, state transition coverage, and synchronization coverage. Requisite support tools include a static concurrency analyzer and either a program transformation system or a powerful run-time scheduler. The techniques proposed are suitable for Ada or CSP-like languages. Best results are obtained for programs having only static naming of tasking objects.

Index Terms—Concurrency, coverage criteria, structural testing, testing criteria hierarchy.

I. INTRODUCTION

Testing of sequential programs has grown to be a fairly sophisticated process, with various methodologies and tools available for use in building and demonstrating confidence in the program being tested. The emergence of concurrent programming in recent years, however, has presented new testing problems and difficulties which cannot be solved by regular sequential program testing techniques. To this end, we present an approach motivated from the structural testing of sequential programs that is analogous to coverage of nodes, edges, and paths of a program flowgraph. Traditional structural (program-based) testing is briefly reviewed, then analogous concepts useful in the testing of concurrent programs are presented. As is true with sequential programming, the structural testing approach to concurrent programs is not offered as a complete methodology, but only a part. With these beginnings, further methods can be developed to attain more sophisticated methods. The main contribution is that of a starting point.

Metrics for the structural testing of a sequential program can be defined with respect to a flowgraph model of a program. In this graph, each node represents a statement or a collection of sequential statements that would be executed as a block. The edges between the nodes represent the flow of control from one block of code to the next. A node with multiple exiting edges represents a branch predicate or a code segment with a branch predicate as the last statement [2], [3].

With respect to this graph, various forms of control-flow and data-flow testing metrics can be defined. As examples, three well-established control-flow criteria of structural testing are path testing, branch testing, and statement testing. The strongest data-flow path selection criteria is all-DU-paths (definition-use), which requires every definition-clear subpath from every definition to all the successor nodes of each use reached by that definition that is a simple cycle or cycle-free to be tested. As indicated by Clarke et al. [4], there is a hierarchy among these and the other data-flow path selection criteria regarding the effectiveness of testing based on the various criteria. Path testing (all-paths criterion) is clearly the most extensive, with the most conclusive test results. It is also the most impractical criterion from the standpoint of requiring an infinite (in most cases) number of paths to be tested. The next step down in the hierarchy is the data-flow path criterion, all-DU-paths. Weyuker has demonstrated that this may represent a practical goal for well-written code [5]. At the bottom of the hierarchy are branch testing (all-edges) and statement testing (all-nodes); these are the least effective methods, but are offered as a base point with a finite test set.

The "lower" methods are ineffective in revealing all problems, leaving some hidden in the code. Therefore structural testing of sequential programs in and of itself is not a complete testing method. It should only be a part of a more comprehensive testing methodology [6].

Concurrent Programs

The concurrent programs that we are interested in testing include those written in languages such as Ada and CSP. Common to these languages, and several others, are explicit identification of large grain parallel computation units, explicit synchronization via a rendezvous-style mechanism, and some unpredictable scheduling activity by a run-time supervisor.

Unfortunately, there is an almost total lack of rigorous, organized approaches to the testing of programs written in such languages. Techniques designed for application to sequential programming can be applied, of course, but they fail to directly address the testing problems peculiar to concurrent programs. The scheduler presents one key difficulty. A program can be supplied the same set of test data on two different executions, yet exhibit markedly different behavior. This behavior represents the effects of the scheduler making different choices in response to conditions external to the program, such as the load on a time-shared machine. Moreover, if the program is ported to a new execution environment, an entirely different scheduler may be present. Differences in scheduler algorithm are permit-
ted by the Ada standard, for example. Similar comments apply to "nondeterministic" statements. The consequence, of course, is that the tester of a concurrent program must contend with a variety of new difficulties.

Many issues of correctness can and should be addressed before development proceeds to the testing stage. Static analysis, formal verification, and symbolic execution techniques are in this category. Nevertheless, our focus will be on the testing issue. The method presented below of structural testing for concurrent programs is far from perfect, however, and similar to structural techniques for sequential programs, will leave some errors hidden. Other methodologies will be required to detect them. For the moment, though, it seems justified to bridge an admittedly weak sequential program testing method to concurrent programming. From this starting point, it is hoped that valuable and stronger concurrent testing methods will be developed.

Concurrency states and related notions are defined in the following section. Structural coverage criteria for concurrent programs are presented in Section III; the hierarchical relationships between them are proven in Section IV. An example of the application of the criteria is the focus of Section V. Expected limits of reachability analysis in practice are presented in Section VI. Uses of the metrics are discussed in Section VII. Section VIII compares related work, and Section IX presents conclusions.

II. CONCURRENCY STATES

The structural testing metrics for concurrent programs are defined with respect to concurrency states. The notion has appeared in many places, including static analysis of concurrent programs [7], [8] and dynamic monitoring for errors in concurrent programming [9]. Perhaps its most well-known manifestation is in the reachability analysis of Petri nets [10]. To provide focus, the discussion in the remainder of this paper will be with respect to Ada. Accordingly, the concurrency state notion is drawn from [8].

The technique can be defined with respect to a graph model of programs. In particular, each Ada program unit (subprogram, task, package, or generic) defines a flowgraph: each statement in the unit is represented by a node in the graph; each transfer of control is represented by a directed edge. A path through the graph represents a sequence of statements. Not all paths through the graph correspond to executable sequences of statements, of course. Determination of the executable paths requires examination of the program logic—the semantics of the branch conditions, etc. This determination is in the realm of symbolic execution or formal verification.

Informally, each concurrency state displays the next synchronization-related activity to occur in each of a system’s tasks. A legal sequence of states presents a history of synchronization activities for a class of program executions.

A successor relationship exists between concurrency states. Let C be a concurrency state for program S. Informally, the successors of a concurrency state are all concurrency states that may follow occurrence of C in some execution. Let C' be a member of succ(C). If state C arises during execution of program S, then with respect to synchronization activities, it is possible for S to directly progress to state C'. At least one task in the system must advance to a successor node; no task may advance further than that. Advancement to C' may involve the activation or termination of a task.

Now some concepts related to sequences of concurrency states are defined. A concurrency history is a sequence of concurrency states, where each element of the sequence Ck is a member of succ(Ck-1), for all k ≥ 1. A concurrency history begins with the initial state of a system S—namely, (elaborate-declarations-of-main-program, “inactive”, . . . , “inactive”). A concurrency history is thus a sequence of snapshots of the execution of S, starting with system invocation. A proper concurrency history is a finite concurrency history of which all elements are unique, save possibly the final element of the sequence. That is, “loops” in the concurrency history are prohibited. The complete concurrency history of a program S is the set of all proper concurrency histories of S. The complete concurrency history can be represented as a graph, with each node representing a unique concurrency state and each edge representing a transition from one concurrency state to the other. Once the concurrency state graph has been constructed, a concurrency history is a sequence of concurrency states lying on some path within the concurrency state graph structure.

III. COVERAGE METRICS FOR CONCURRENT PROGRAMS

Using the concurrency state graph described in Section II, coverage criteria can be defined analogously to those defined for sequential testing.1 The following notation is used, in addition to that introduced in the preceding section. C and C' are concurrency states, and H(S) is the complete concurrency history, which is the set of all possible proper histories. Additionally, E is defined as the set of all pairs (C, C') such that C' ∈ succ(C).

In the following metric definitions we define a criterion as a predicate that assigns a truth value to a pair (S, P), where S is a program and P is a subset of all the paths through the concurrency graph. Then we can say that a pair (S, P) satisfies a criterion K iff K(S, P) = true [4].

Establishing a hierarchy of criteria, the all-concurrency-paths criterion is at the top. This criterion requires that every path through the concurrency graph be covered by the test set, including those paths of infinite length. Clearly, this is impractical but is included for completeness, as is done in discussions of testing of sequential programs.

Criterion 1

The pair (S, P) satisfies the all-concurrency-paths criterion iff P = (all paths through the concurrency graph).

1The fundamental difference between the sequential criteria and concurrency criteria should be apparent now: the sequential criteria are with respect to a model of the program under test, the concurrency criteria are with respect to the execution state space of the program under test. Thus while the criteria can be defined analogously, there is a vast difference between the bases for the criteria.
A second and more reasonable criterion, all-proper-cc-histories, examines \(H(S)\), the set of all possible proper concurrency histories. Here we are only interested in paths through the concurrency graph of finite length that have no duplicated states on the path.

**Criterion 2**

The pair \((S, P)\) satisfies the all-proper-cc-histories criterion iff \(P = H(S)\), where \(H(S)\) is the set of all finite concurrency histories in the graph, beginning with the initial state of the system, and with no loops and no duplicated states, with the exception of the final elements of the sequences.

Two criteria that might be more reasonable to accomplish are: all-edges-between-cc-states and all-cc-states. All-edges-between-cc-states is our weakest state transition coverage criterion and is analogous to branch testing of sequential program testing. It tests every edge in the graph that transfers control from one node to its successor. The all-cc-states concurrency state coverage criterion is analogous to sequential program statement testing, in that it requires that every node (representing a concurrency state) in the concurrency graph be tested.

**Criterion 3**

The pair \((S, P)\) satisfies the all-edges-between-cc-states criterion iff for all edges \((C, C') \in E\) there is at least one path in \(P\) along which \((C, C')\) occurs.

**Criterion 4**

The pair \((S, P)\) satisfies the all-cc-states criterion iff for all concurrency states \(C\) there is at least one path in \(P\) along which \(C\) occurs.

As discussed in Section VI, the number of unique concurrency states may be expected to grow exponentially with the number of tasks in a program. This argues that the preceding criteria could prove to be impractical. Therefore it is necessary to step down another level in the hierarchy and seek a criterion with a more narrow focus. One approach is to pick states that are interesting or meritorious of closer examination. Typically, these nodes would have greater potential for errors to occur or would correspond to a major tasking-related activity. As an example, consider an all-possible-rendezvous criterion. This criterion would label as interesting those nodes that indicate rendezvous occurring between tasks.

**Criterion 5**

The pair \((S, P)\) satisfies the all-possible-rendezvous criterion iff for all concurrency states \(C\) that involve a rendezvous with another task there is at least one path in \(P\) along which \(C\) occurs.

The selection of criteria at this level in the hierarchy provides a realistic metric with which to work. While not as strong as higher criteria, they would offer a reasonable basis for actual executions of the program over a set of test data. Moreover, one has the intuitive feel that the significant tasking-related portions of the program have been exercised.

### IV. COVERAGE CRITERIA HIERARCHY

The coverage criteria hierarchy proposed in Section III and shown in Fig. 1 is justified in this section. First, some definitions of terms used in the theorems are presented.

Criterion A subsumes criterion B if any set of paths that satisfies criterion A also satisfies criterion B.

Criterion A strictly subsumes criterion B if criterion A subsumes criterion B, but criterion B does not subsume criterion A.

Criterion A and criterion B are incomparable if criterion A does not subsume criterion B, and criterion B does not subsume criterion A.

The notation \((mp(n))\) denotes a path \(p\) in the concurrency state graph from node \(m\) to node \(n\), where every node represents a concurrency state. The path \(p\) can be of length 0 when \((m, m)\) is itself \(E\), where \(E\) is the set of all edges in the concurrency state graph. The path \(p\) can also be expressed as a sequence of ordered nodes \(m, q_1, q_2, \ldots, q_k, n\), where \((m, q_1), (q_i, n)\) and \((q_i, q_{i+1}) \in E\) for all \(i, 1 \leq i < k\).

\(N\) is the set of all nodes in the concurrency state graph. The start node of the program is represented as \(n_s\), and \(n_t\) represents the set of terminal nodes. A path is a complete path when it begins at the start node \(n_s\) and ends at a terminal node \(n_t\).

**Theorem 1**

All-concurrency-paths strictly subsumes all-proper-cc-histories.

Let \(P_1\) be the set of all paths through the concurrency state graph of program \(S\), some of which may be infinite in length; \((S, P_1)\) satisfies the all-concurrency-paths criterion. All-concurrency-paths includes all paths through the graph that follow backward pointers (loops), possibly more than once. Now let \(P_2\) be the set of all paths \((mp(n))\), where \(m = n_s, n \in N\), and \(p\) is a sequence of ordered nodes \(m, q_1, q_2, \ldots, q_k, n\), where \((m, q_1), (q_i, n)\) and \((q_i, q_{i+1}) \in E\) for all \(i, 1 \leq i < k\), such that all nodes \(q_i\) and \(q_j\) are unique for \(i \neq j\), where \(j < k\) (the last node need not be unique). By definition, \((S, P_2)\) satisfies the all-proper-cc-histories criterion. However, any path \(n_s(p)n_t \in P_2\) satisfying the all-proper-cc-histories criterion would terminate at the first cc-state node to
be repeated, designated here as \( T \). There exists at least one path in \( P \) that, if terminated at \( n_r \), is identical to \( n_r(p)n_r \).

Since all-proper-cc histories are covered by these extended all-proper-cc-histories paths, all-concurrency-paths also satisfies all-proper-cc-histories. Therefore all-concurrency-paths subsumes all-proper-cc-histories.

Now note that \((S, P_2)\) does not satisfy the all-concurrency-paths criterion because it does not have any paths with loops, so all-proper-cc-histories does not subsume all-concurrency-paths.

Therefore all-concurrency-paths strictly subsumes all-proper-cc-histories.

**Theorem 2**

All-proper-cc-histories strictly subsumes all-edges-between-cc-states.

Let \( P_1 \) be the set of all paths \((m)p(n)\) through the concurrency state graph of program \( S \), where \( m = n_1, n \in N \), and \( p \) is a sequence of ordered nodes as before, where \((m, q_1), (q_k, q_{k+1}) \in E\) for all \( i \leq j \leq k \), such that all nodes \( q_i \) and \( q_j \) are unique for \( i \neq j \), where \( j \leq k \). \((S, P_1)\) satisfies the all-proper-cc-histories criterion by definition. \((S, P_1)\) also satisfies the all-edges-between-cc-states criterion because all of the edges \((m, q_1), (q_k, q_{k+1}) \in E\), \( 1 \leq i \leq k \) are necessarily included in \( P_1 \). Thus all-proper-cc-histories subsumes all-edges-between-cc-states.

Conversely, let \( P_2 \) be a set of paths \((m)p(n)\) such that all edges \( E \) of the concurrency state space of program \( S \) are represented in \( P_2 \). Then \((S, P_2)\) satisfies the all-edges-between-cc-states criterion. However, \( P_2 \) can be chosen in such a way that all edges are represented without using all paths, and the all-proper-cc-histories criterion cannot be satisfied. For example, in Fig. 2, a program and the set of paths that includes only paths \((1, 3)\) and \((2, 5, 4)\) satisfy the all-edges-between-cc-states for the graph. However, all-proper-cc-histories is not satisfied by this program and set, because paths \((1, 5, 4)\) and \((2, 5, 3)\) are not included in the set, although they represent distinct proper concurrency histories. Thus all-edges-between-cc-states does not subsume all-proper-cc-histories.

Therefore all-proper-cc-histories strictly subsumes all-edges-between-cc-states.

**Theorem 3**

All-edges-between-cc-states strictly subsumes all-cc-states.

Choose \( P_3 \) for program \( S \) such that for all paths \((m)p(n)\), all edges \( E \) are represented in \( P_3 \). \((S, P_3)\) satisfies the all-edges-between-cc-states criterion. It also satisfies the all-cc-states criterion because each node appears on at least one of the edges \( E \). Each state \( C_k \) in a concurrency history is a member of \( sucE(C_{k-1}) \), for all \( k > 1 \); thus there must be an edge \((C_{k-1}, C_k) \in E\). Therefore all edges between-cc-states subsumes all-cc-states.

Now it is possible to choose \( P_2 \) such that all nodes occur in at least one path \((m)p(n)\) in \( P_2 \), satisfying the all-cc-states criterion. However, it is possible that \( P_2 \) does not contain a path from a node \( q_i \) to a successor \( q_{i+1} \), but \((q_i, q_{i+1}) \in E \). In Fig. 3, a program and the set of paths consisting of only \((1, 2)\) and \((2, 3)\) satisfies the all-cc-states criterion. However, \((1, 3)\) is not included in this set; all-edges-between-cc-states is not satisfied. All-cc-states does not subsume all-edges-between-cc-states.

Therefore all-edges-between-cc-states strictly subsumes all-cc-states.

**Theorem 4**

All-cc-states strictly subsumes all-possible-rendezvous.

Let \( P_2 \) be any set of paths \((m)p(n)\) such that all concurrency states of program \( S \) are represented along at least one path in \( P_2 \), satisfying the all-cc-states criterion. Since the set of rendezvous states is a proper subset of all states in \( S \), \((S, P_1)\) will also satisfy the all-possible-rendezvous criterion, and all-cc-states subsumes all-possible-rendezvous.

Now choose \( P_3 \) to be some set of paths \((m)p(n)\) such that each rendezvous state occurs on at least one of the paths. \((S, P_3)\) satisfies the all-possible-rendezvous criterion, because each rendezvous state is represented in \( P_3 \). The all-cc-states criterion, however, is not necessarily satisfied because \( P_3 \) can be chosen in such a way that excludes some states in the graph; for example, those representing a task activation or termination.

Therefore all-cc-states strictly subsumes all-possible-rendezvous.

**V. AN EXAMPLE: DINING PHILOSOPHERS**

As further clarification of the ideas presented above, the method will be informally applied to the dining philosophers problem. Deadlock occurs in our particular formulation of the problem when every philosopher holds the left fork, but blocks
procedure DINING-PHILOSOPHERS is
  NUMBER_OF_SEATS : constant := 5;
type SEAT_ASSIGNMENT is range 1 .. NUMBER_OF_SEATS;
task type FORK is
    entry UP;
    entry DOWN;
  end FORK;
  FORKS : array (SEAT_ASSIGNMENT) of FORK;
  task body FORK is
    begin
      loop
        accept UP;
        accept DOWN;
      end loop;
    end FORK;
  generic
    N : in SEAT_ASSIGNMENT;
  package PHILOSOPHER is
    -- used to identify each philosopher task
  end PHILOSOPHER;
  package body PHILOSOPHER is
    EAT, THINK : constant := 10.0;  -- seconds
    task P;
    task body P is
      begin
        loop
          FORKS(N).UP;  -- acquire left fork
          FORKS(N mod NUMBER_OF_SEATS + 1).UP;  -- and right fork
          delay EAT;
          FORKS(N).DOWN;  -- put down left fork
          FORKS(N mod NUMBER_OF_SEATS + 1).DOWN;  -- and right fork
          delay THINK;
        end loop;
      end P;
    end PHILOSOPHER;
    package Aquinas is new PHILOSOPHER(1);
    package Bonhoeffer is new PHILOSOPHER(2);
    package Kierkegaard is new PHILOSOPHER(3);
    package Schaeffer is new PHILOSOPHER(4);
    package Tilich is new PHILOSOPHER(5);
    begin
      null;
    end DINING-PHILOSOPHERS;
end DINING-PHILOSOPHERS;

Fig. 4. Dining philosophers example.

on the entry call to pick up the right fork. The Ada program
in Fig. 4 is written for five philosophers.\(^3\)

This program would generate 11 flowgraphs—one for each
fork task, one for each philosopher, and one for the main
program. For the sake of simplicity, the remainder of the
example will use only two philosophers, which requires only
five program subunits, as shown in Fig. 5.

Now the concurrency states can be generated (Table I).
Identifying the possible next concurrency states establishes the
relationship between concurrency states that allows construc-
tion of a concurrency graph.

Representing each concurrency state as a node in a graph
and drawing arcs from a concurrency state to each of its
possible next concurrency states results in the concurrency
diagram, Fig. 6. This graph is analyzed with respect to the
coverage criteria.

With respect to the concurrency graph, Examples 1 through
3 illustrate aspects of Theorems 1 through 3, respectively. For
these examples, \(P_A\) refers to the set of all paths through
the graph; \(P_H\) is the set of all paths satisfying the
all-proper-cc-

\(^3\)Each philosopher task uses the generic value parameter, which is defined
upon instantiation, to identify itself.

Fig. 5. Flowgraphs.

Example 1
\((m)_{P_1}(n) = (1,2,3,4,5,6,7)\) is \(\in P_A\) and also \(\in P_H\). \(P_H\)
Table I
Concurrency states, optimized representation, as discussed in Section VI

<table>
<thead>
<tr>
<th>Concurrency State No.</th>
<th>Fork(1)</th>
<th>Fork(2)</th>
<th>Philos(1)</th>
<th>Philos(2)</th>
<th>Next States</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>accept Up</td>
<td>accept Up</td>
<td>Fork(1).Up</td>
<td>Fork(2).Up</td>
<td>2.8</td>
</tr>
<tr>
<td>2</td>
<td>accept Down</td>
<td>accept Up</td>
<td>Fork(2).Up</td>
<td>Fork(2).Up</td>
<td>3.7</td>
</tr>
<tr>
<td>3</td>
<td>accept Down</td>
<td>accept Down</td>
<td>Fork(1).Down</td>
<td>Fork(2).Up</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>accept Up</td>
<td>accept Down</td>
<td>Fork(2).Down</td>
<td>Fork(2).Up</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>accept Down</td>
<td>accept Up</td>
<td>Fork(2).Up</td>
<td>Fork(2).Up</td>
<td>3.7</td>
</tr>
<tr>
<td>7</td>
<td>accept Down</td>
<td>accept Down</td>
<td>Fork(2).Up</td>
<td>Fork(1).Up</td>
<td>7.9</td>
</tr>
<tr>
<td>8</td>
<td>accept Up</td>
<td>accept Down</td>
<td>Fork(1).Up</td>
<td>Fork(1).Up</td>
<td>7.14</td>
</tr>
<tr>
<td>9</td>
<td>accept Down</td>
<td>accept Up</td>
<td>Fork(1).Up</td>
<td>Fork(2).Down</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>accept Down</td>
<td>accept Up</td>
<td>Fork(1).Up</td>
<td>Fork(1).Down</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>accept Up</td>
<td>accept Down</td>
<td>Fork(1).Up</td>
<td>Fork(1).Up</td>
<td>7.9</td>
</tr>
<tr>
<td>13</td>
<td>accept Up</td>
<td>accept Down</td>
<td>Fork(1).Up</td>
<td>Fork(1).Up</td>
<td>7.14</td>
</tr>
<tr>
<td>14</td>
<td>accept Down</td>
<td>accept Up</td>
<td>Fork(1).Up</td>
<td>Fork(2).Down</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>accept Down</td>
<td>accept Up</td>
<td>Fork(1).Up</td>
<td>Fork(1).Down</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>accept Up</td>
<td>accept Down</td>
<td>Fork(2).Up</td>
<td>Fork(2).Up</td>
<td>7.18</td>
</tr>
<tr>
<td>18</td>
<td>accept Down</td>
<td>accept Down</td>
<td>Fork(1).Down</td>
<td>Fork(2).Up</td>
<td>19</td>
</tr>
<tr>
<td>19</td>
<td>accept Up</td>
<td>accept Down</td>
<td>Fork(2).Down</td>
<td>Fork(2).Up</td>
<td>16</td>
</tr>
</tbody>
</table>

satisfies the all-proper-cc-histories criterion, but not the all-concurrency-paths criterion, because it does not contain paths in $P_A$ with loops such as $(m)p_7(n) = (1, 2, 3, 4, 5, 6, 3, 4, 5, 6, 7)$.

Example 2

$P_E$ satisfies the all-edges-between-cc-states criterion, but not the all-proper-cc-histories criterion, because it does not necessarily contain paths in $P_H$ that start at $n_s$, such as $(m)p_4(n)$ from Example 1. This path may be replaced in $P_E$ with, for example, the two paths $(m)p_3(n) = (5, 6, 7)$ and $(m)p_4(n) = (1, 2, 3, 4, 5)$.

Example 3

$P_{CC}$ satisfies the all-cc-states criterion, but not the all-edges-between-cc-states criterion, because it does not necessarily contain edges on all of the paths in $P_E$. Nodes 3 and 6, among others, occur on $(m)p_1(n)$ from Example 1. However, edge (6, 3) does not occur on this path, and nodes 3 and 6 need not occur on any other path in $P_{CC}$.

VI. PRACTICAL ISSUES

Static concurrency analysis techniques based on reachability analysis are limited in practice by state space explosion. Analysis results give us a rough idea of this limit in terms of the size of the concurrency graph and the time required to construct it. And if the size of the concurrency graph of an arbitrary program can be estimated, then the program's suitability for static concurrency analysis could be determined.

The sizes and times to construct the concurrency graphs for several variations (of deadlock prevention approaches) of the dining philosophers were initially presented in [11]. The CATS tool suite has subsequently been used to analyze additional dining philosophers programs, incorporating the several optimizations described below; the results are shown in Table II. The execution times are for construction of the concurrency graph; the times required for deadlock checking and temporal logic assertion checking are typically small compared to those for the graph construction. These results illustrate the expectation that the concurrency graph size is exponential in the number of tasks.

The dining philosophers is of interest because the predominant task interactions result in such a large concurrency state space. To demonstrate the feasibility of static concurrency analysis for more typical programs, the tasking structure of the Chiron-1 user-interface development system [12] was analyzed using CATS. The Chiron design separates it into two operating system processes: the server and the client/application—our

The approximate execution times in Tables II and III are for concurrency state graph construction on a Sun 4/400 with VeriX Ada 6.0g. A Sun SparcStation 2 offered similar performance.
TABLE II
CONCURRENCY GRAPH CONSTRUCTION PERFORMANCE: DINING PHILOSOPHERS

<table>
<thead>
<tr>
<th>Philosophers</th>
<th>Tasks</th>
<th>States</th>
<th>Edges</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>19</td>
<td>28</td>
<td>2.4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>84</td>
<td>186</td>
<td>2.8</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>375</td>
<td>1112</td>
<td>3.5</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>1653</td>
<td>6130</td>
<td>5.4</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>7282</td>
<td>32412</td>
<td>21.6</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>32063</td>
<td>166502</td>
<td>648</td>
</tr>
</tbody>
</table>

TABLE III
CONCURRENCY GRAPH CONSTRUCTION PERFORMANCE: CHIRON—1.0

<table>
<thead>
<tr>
<th>Component</th>
<th>Tasks</th>
<th>States</th>
<th>Edges</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>server</td>
<td>8</td>
<td>2624</td>
<td>7782</td>
<td>8.0</td>
</tr>
<tr>
<td>client/application</td>
<td>12</td>
<td>29625</td>
<td>140701</td>
<td>376</td>
</tr>
</tbody>
</table>

analysis was partitioned along this split. As shown in Table III, the concurrency state graphs are of a very manageable size.

A. Optimized Concurrency Graphs

A high-payoff approach to reducing the size of the concurrency graph is to optimize the constituent task flowgraphs. While rendezvous is in general represented by three nodes, Long and Clarke note that this can be reduced to two for simple synchronization, with no accept body [13]. The three nodes surround two edges, representing start and end of the rendezvous. The optimized flowgraph represents the synchronization as a single edge. Delay statements can be ignored if they do not affect the order of task interactions—just the time between them. These optimizations are applied in the dining philosophers analysis described above.

The concurrency graph itself may be simplified by ignoring inconsequential task activity such as task activation and termination in service tasks containing infinite loops. Because the main task of the dining philosophers performs no visible processing, it is omitted from the concurrency graph.

B. Concurrency Graph-Size Estimation

Wampler implemented Taylor's algorithm for constructing the concurrency state graph and demonstrated its application to the dining philosophers problem [14], [8]. In this example, the number of unique concurrency states is approximately equal to \( \binom{n}{t} \cdot \frac{1}{2} \), where \( n \) = total number of state nodes in all tasks, \( t \) = total number of tasks in the system, and \( n/t \) is the average number of state nodes per task. This provides a basis for estimating the size of the state space for similar programs—those with rendezvous comprising nearly all of the tasking activity.

The total number of state nodes \( n \) is best determined by inspection of the flowgraph of each task. Alternatively, \( n \) can be roughly estimated by inspection of the source code. Each feature listed in Table IV contributes to the state node count as indicated. The count for a select statement is the sum of the node counts in all of its branches. If the node count in each branch is the same, then it is just the product of the branch count and the number of branches.

While this process is predominantly mechanical, dynamic task creation clouds the estimate. If tasks are dynamically created during execution, it may not be possible to determine the number of tasks created prior to execution, which is why some static analysis techniques avoid dynamic tasks entirely. Similarly, task arrays should be handled carefully.

VII. USING THE COVERAGE CRITERIA

Given the preceding criteria, it remains to consider how they can be employed effectively. The primary concerns are: (i) determining what kind of coverage is achieved with a set of test executions, (ii) developing test data to increase the amount of coverage, and (iii) addressing the run-time scheduler issue such that the execution of specified concurrency paths can be guaranteed, thereby increasing the coverage obtained (or maximizing the effect—coverage—of a given test suite).

A. Measuring Coverage

It is quite clear that structural coverage metrics, whether for sequential or concurrent programs, are only useful when considered with respect to a set of test executions. A single run will not generate complete coverage for any metric except on the most trivial of programs. Thus coverage statistics need to be retained and correlated over a set of runs. Three reasonable techniques for noting and recording the concurrency
states encountered in a single run are program transformation, specialized run-time systems, and symbolic evaluation.

The relevant program transformation approach has been pioneered by German et al. [15], [9]. In the context of monitoring for tasking errors, they have developed techniques for transforming an Ada tasking program \( P \) into a program \( P' \), such that \( P' \) exhibits all the behavior of \( P \), but, in addition, \( P' \) records the state transitions through which the program progresses. The transformation adds a monitoring task to \( P \) and many entry calls from the existing tasks to the monitor such that all tasking activities of \( P \) are duly recorded. If an error arises during execution, this monitoring information can be used to aid debugging. If no error arises, however, the transitions can be stored away for assessing test coverage.

The drawback to the transformation approach is apparent, however. An overhead is incurred, arising from the insertion of the extra entry calls and the monitor task. Moreover, as shown by Gait and noted by Helmbold and Luckham, this so-called probe effect can almost completely mask synchronization errors if the overhead is large [16]. An alternative implementation technique is to modify the run-time scheduler itself to directly record the requisite information; or symbolic evaluation may be used to assess worst-case coverage for a known input data set.

B. Generating Test Data

Given the coverage results of a set of test data applied to a program with a particular criterion, the tester can determine whether certain “interesting” states are yet to be covered. If there are some, it remains to generate the additional data necessary. Beyond providing the basis for the coverage metrics, the concurrency histories provide a guide to the development of these test cases [17]. Using the concurrency graph and a tool to do depth first search on the graph, for example, a partial concurrency history to the desired state can be generated. This concurrency history can then be related to (set of) paths in the flowgraphs of the original program, and thus to a set of predicates constraining the program’s inputs to cause the path to be executed. (Problems with the scheduler are considered below.) This is at least familiar ground, if not easy. A symbolic executor could be applied to the partial concurrency history to collect these predicates [18], [19].

C. Forcing a Path

One of the difficulties encountered in the testing of concurrent software is the control of execution. In any particular execution there may, for example, be more than one open (accept) alternative in a select statement. The scheduler may nonpredictably choose one of the open alternatives and initiate a rendezvous accordingly. If an error is discovered when a particular concurrency path is followed on a particular set of test data, the tester would like to be able to repeat the identical execution by forcing selection of the same rendezvous that followed the tested concurrency path on the same data previously. Alternatively, if it is known that a state has not yet been covered and data is provided to reach that state if a certain rendezvous is chosen, it would be helpful to be able to guarantee that selection. This raises the need for a controllable scheduler that allows the user to choose the specific rendezvous to be followed. In particular, the scheduler should accept the specification of a path through the concurrency state graph and make its transitions in accordance with the transitions indicated by the path. (Of course, if the test data dictates another path be followed, then an error in the testing procedure should be reported.) This issue has also been raised by Brindle et al. [20], who discuss the design for a tasking debugger.

Tai et al. [21] have developed an approach to reproducing the entry call arrival and rendezvous sequencing of an Ada program using an added task to control the execution sequence, similar to the transformation of German et al. Every tasking-related action must request permission from the controlling task before proceeding; any feasible sequence can therefore be enforced. This approach lends itself to automation and does not require modifications to the scheduler. LeBlanc and Mellon-Crummy describe a method, termed Instant Replay, for reproducing execution behavior that is especially suited to tightly coupled systems [22]. During execution, the relative order of significant events is saved, but not the data, so the approach requires less time and space than other methods. Replay then uses this history and the input data set.

VIII. RELATED WORK

Our structural testing metrics hide purely sequential activity in each concurrency state; we then explore the reachability graph in search of synchronization anomalies. An “opposite” approach represents a concurrent program as a set of sequential programs, as suggested by Weiss [23]. The methods and theory of testing sequential programs can then be extended to concurrent programs by applying them to the members of the set. For any sequential-program-based criterion and fixed input there is consequently a hierarchy of concurrency-program-based adequacy criteria which corresponds to the extent of coverage of the serializations.

Morasca and Pezzè propose test adequacy criteria based on the coverage measures of Petri net topology [24]. A test datum is defined as a particular execution (firing sequence) in addition to the input/output data. Criteria based on firing coverage as well as those based on transition coverage are defined; the former are for a particular input/output data set, while the latter are regardless of data. Comparison of the Petri net-based adequacy criteria with those based on the concurrency state graph is complicated by the different bases of the two coverages—the state transitions and the concurrency state space. Nevertheless, the all-concurrency-paths criterion roughly corresponds to the transition-sequence testing criterion, which is satisfied by a test (set of firing sequences) containing all transition sequences that can be extracted from the firing sequences of the net. Similarly, the all-edges criterion approximately corresponds to the transition testing criterion. All-proper-ce-histories loosely corresponds to \( n \)-times testing, although the former requires that each proper history be included at least once in the set of paths tested, while the latter places a limit on the number of firings of each transition to \( n \).
IX. CONCLUSION

This paper has outlined several metrics on which a structural testing methodology for concurrent programs can be based. No claim is made that the methodology suggested is ideal; rather this presentation is offered as a starting point for further work in the area. Clearly, it is impractical at present to analyze a large program by building its concurrency graph. Analysis techniques based on process algebra offer the possibility of hierarchically composing partial analysis results [25].

The methodology presented is applicable to programs written in languages such as CSP and Ada. Since the technique rests heavily on information derived from static analysis, best results are likely to be obtained with languages supporting static identification of tasking-related entities.

Perhaps the most interesting aspects of this work are the implications it has for the structure of tools for assessing concurrent programs (and thus for a plan to conduct empirical evaluation). This testing methodology requires cooperative application of several sophisticated tools—among them a static concurrency analyzer, a coverage assessment utility, and a "programmable" run-time scheduler enabling check-out of specific synchronization behavior. This in turn strongly argues that the tools should be composed of small modular components and be housed in a programming environment that shields the user from the complexity of the internal structure of the tools. This complexity seems to be endemic to support of concurrent programs, unfortunately.

All work in this area must take cognizance of the context in which testing tools are used. When errors are discovered, the program is subsequently corrected. Unless the testing and analysis tools are thoughtfully constructed, potentially massive amounts of recomputation will be performed to generate the new basis of static analysis information. It is essential that this be avoided. Moreover, some previous test runs and therefore coverage data may still be relevant. Thus the "re-analysis" and "re-test" problems deserve significant attention.

The CATS approach is an example of a tool suite developed to automate the processes of building the concurrency graph and its related histories [11]. Executing the program over a set of test data and gathering information for applying one or more criterion to the results of the executions in order to expand the test data set increases confidence in the program. An integrated application of tools through a software development environment is illustrated by the construction of CATS atop Arcadia [26].

REFERENCES


Richard N. Taylor was born in Denver, CO, in 1952. He received the B.S. degree in applied mathematics and the M.S. and Ph.D. degrees in computer science from the University of Colorado, Boulder, in 1974, 1976, and 1980, respectively.

From 1977-1979 he was with Boeing Computer Services in Seattle. In 1981-82 he was an Assistant Professor of Computer Science at the University of Victoria, Victoria, BC, Can. Since 1982 he has been with the Department of Information and Computer Science at the University of California, Irvine, where he is a Professor. His research interests are centered on methods...
for effectively developing reliable software systems. These interests include techniques for analyzing, testing, and debugging concurrent programs. He is also working in the area of environments supporting software development and maintenance processes, emphasizing graphical user interfaces, team-level issues, and extensibility.

Professor Taylor was a 1985 recipient of a Presidential Young Investigator Award.

David L. Levine received the B.S.M.E. degree from Cornell University in 1978, the M.S.C.S. degree from George Washington University in 1987, and is working toward the Ph.D. degree in computer science at the University of California, Irvine.

His industry experience includes advanced automotive drivetrain simulation with the Chrysler Corporation and The Aerospace Corporation. His research interests are in software engineering, particularly the design and analysis of concurrent programs.

Cheryl D. Kelly received the B.S. degree in computer science from California State University, Fullerton, and the M.S. degree in information and computer science from the University of California, Irvine.

She is a member of the technical staff at the Jet Propulsion Laboratory, Pasadena, CA, where she has worked on software metrics definition and data collection within the software process, and is currently a Systems Analyst in the Problem/Failure Operation Center.