Polytechnic
UNIVERSITY
Brooklyn · Long Island · Westchester

Toward Synergy of Finite State Verification And Testing

Gleb Naumovich Phyllis G. Frankl

Department of Computer and Information Science

Technical Report
TR-CIS-2000-04
06/21/2000

Toward Synergy of Finite State Verification and Testing

Gleb Naumovich
Phyllis G. Frankl *
Polytechnic University
6 Metrotech Center
Brooklyn, NY 11201

{gleb, pfrankl}@duke.poly.edu

Abstract

Finite state verification (FSV) and testing are usually viewed as competing approaches to software validation. In this short paper, we propose a technique for combining FSV synergistically with testing, with the goal of identifying faults more quickly and with less manual effort than with FSV alone and more effectively than with testing alone. We propose using information about potential faults obtained during the FSV analysis to direct selection, execution, and checking of test data, with the intent of confirming these faults.

1 Introduction

In finite state verification, a finite model of the system is constructed, usually abstracting away many details, and the FSV tool (verifier) explores the state space to determine whether a given property P holds. The model is constructed in such a way that if the verifier determines that P holds for the model, then P also holds for all possible executions (and hence, for all possible test data) of the actual system. In this case, there is no need to test the system for the behaviors captured by P. On the other hand, if the verifier finds a violation of P, it may or may not reflect a property violation in the actual system. Such violation is *spurious* if no violation-revealing path through the system model corresponds to a feasible execution of the system. Normally, given a representation of the property violation on the model, the human analyst (or simply *analyst* hereafter) has to decide whether this violation appears spurious, in which case the analyst has to refine the system model, providing more detail, and then re-run the verifier. The additional details may allow the verifier to determine that P is always satisfied or there may still be a violation. In the latter case, the process continues.

*Supported in part by NSF Grant CCR-9870270.

Appears in Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification, pp. 89–94, June 4-5, 2000, Limerick, Ireland.

This incremental approach to FSV has several weaknesses related to the presence of human factor in the verification process. First, this approach relies on the analyst to decide whether a property violation found by the verifier is spurious or not, which is time-consuming and error-prone. Second, the analyst can only review one property violation at a time, while our technique can use information about all found violations at the same time. Finally, if a violation appears feasible, it is important to analyze a real execution of the system that results in this violation, so that the error in the system can be found and removed. Unfortunately, debugging cannot be used until the analyst manually identifies test data that are likely to produce such an execution.

Our proposed technique uses testing, along with model refinement, to address these weaknesses. When a property violation is found by the verifier, the following steps are performed in parallel:

- An automated testing tool uses information developed during the FSV analysis to direct selection, execution, and checking of test data, with the hope of finding data that shows the violation to be real, and
- The analyst refines the model and re-starts the verifier. Note that the analyst only needs to pick a reasonably important aspect of the system to be modeled during the next run of the verifier, without having to worry about whether the violation is spurious.

If the violation is real, testing will sometimes be able to find test data that exhibits this violation in the system. In this case, the parallel FSV session can be stopped and a debugging session with the found test data can be started. If the violation is spurious, thorough testing of relevant parts of the system may help increase confidence that such is the case, but, of course, will never be able to prove it. The gain in this case is that the analyst is able to start a new, more precise, verification session promptly, which helps to speed up the overall process of FSV. Thus, from the point of view of the FSV analyst, this approach saves some manual work; from the point of view of the tester, this approach helps direct

testing effort toward execution paths that are at risk of violating the specification.

In the remainder of this paper, we refine these ideas further, illustrating with a simple example. Section 2 summarizes relevant background on FSV and testing, Section 3 illustrates the technique and discusses some of the issues, and Section 4 concludes.

2 Background

2.1 Background on finite state verification

Conceptually, many FSV approaches represent the system under analysis as a collection of states in which this system can be during its executions and transitions connecting these states. This construct may be created explicitly (e.g. [4,6,12]) or implicitly (e.g. [9,15]). For simplicity, in this paper we use an explicit representation of the state space, although the proposed techniques can be extended for implicit representations.

Consider the example in Figure 1. The two threads of control, `T1` and `T2`, that comprise this system are represented as FSAs in Figure 1(a). State 0 is the start state in both FSAs. The states representing termination of the threads are indicated with double circles. The transitions between the states are labeled with events in the threads to which they correspond. For example, the transition from state 1 to state 2 of thread `T1`, labeled `start T2`, represents thread `T1` starting thread `T2`. Events `a` and `b` represent some events in the threads that are relevant to the property. Square brackets that follow some events represent conditions on when the event is executed. For example, state 2 of thread `T1` represents this thread before executing an `if` statement with predicate $\texttt{x} > 0$. Event `a` appears on the branch of this `if` statement that is executed when the predicate evaluates to true. τ denotes an empty event, representing absence of any events. For example, the τ-based transition from state 2 to state 3 of thread `T1` means that nothing of interest happens on the branch of the `if` statement that is executed when $\texttt{x} \leq 0$. Note that we assume that `x` is a shared variable that is an input to thread `T1` and is not changed by either `T1` or `T2`.

Formally, we can represent an *FSA* as a tuple (S, s_0, Σ, T), where S is the set of states, s_0 is a unique *start* state, Σ is the set of events, and T is the set of *transitions*. We use the notation $s_1 \xrightarrow{e} s_2$ to represent a transition based on event $e \in \Sigma$ from state $s_1 \in S$ to state $s_2 \in S$. A *path* through an FSA on an event sequence $e_1, ..., e_n$ from Σ is a sequence of transitions $s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} ... \xrightarrow{e_n} s_n$.

In this paper, we assume that FSA-based models of the threads of control are derived from the source code for the system. While construction of models based on high-level descriptions is attractive and has been advocated for FSV [11], since testing is used in our approach, we need a direct mapping between the thread models and the executable code for the system.

A *property* about a software system is a representation of either desirable or undesirable behavior of this system. We define properties in terms of the events observed in the system, using FSAs with a special *violation* state v. The violation state is a sink: $\forall e \in \Sigma_P, v \xrightarrow{e} v$. A property is *violated* on an execution that corresponds to the event sequence $p = e_0, e_1, ...$, if the path through the property FSA on this sequence ends in the violation state.

Figure 1(b) shows a property specifying that on no execution of the system can event `b` be observed if by that time event `a` has been observed an odd number of times. For example, the sequence of events `a`, `a`, `a`, `b` corresponds to the path $0 \xrightarrow{\texttt{a}} 1 \xrightarrow{\texttt{a}} 0 \xrightarrow{\texttt{a}} 1 \xrightarrow{\texttt{b}} v$, and so violates this property. Note that events other than `a` and `b` do not affect this property, which means that if, for example, event `start T2` is contained in a sequence of events, it does not change the current state of the property.

A *reachability graph* represents all reachable states of the system, to the extent that this system is modeled by the FSV technique of choice. In our example, each thread of control in the system is modeled with an FSA, so a state of the system can be represented as an ordered collection of FSA states, one for each thread. The reachability graph is a cross-product of the FSAs for all threads. Figure 1(c) contains the reachability graph for our example.

Paths through the reachability graph represent executions of the system. A path in the reachability graph is *executable* if it corresponds to a real execution of the system. All other paths are *spurious*. If there is a path from the start state of the reachability graph to some state s, such that the property is violated on this path, s is called a *violation* state.

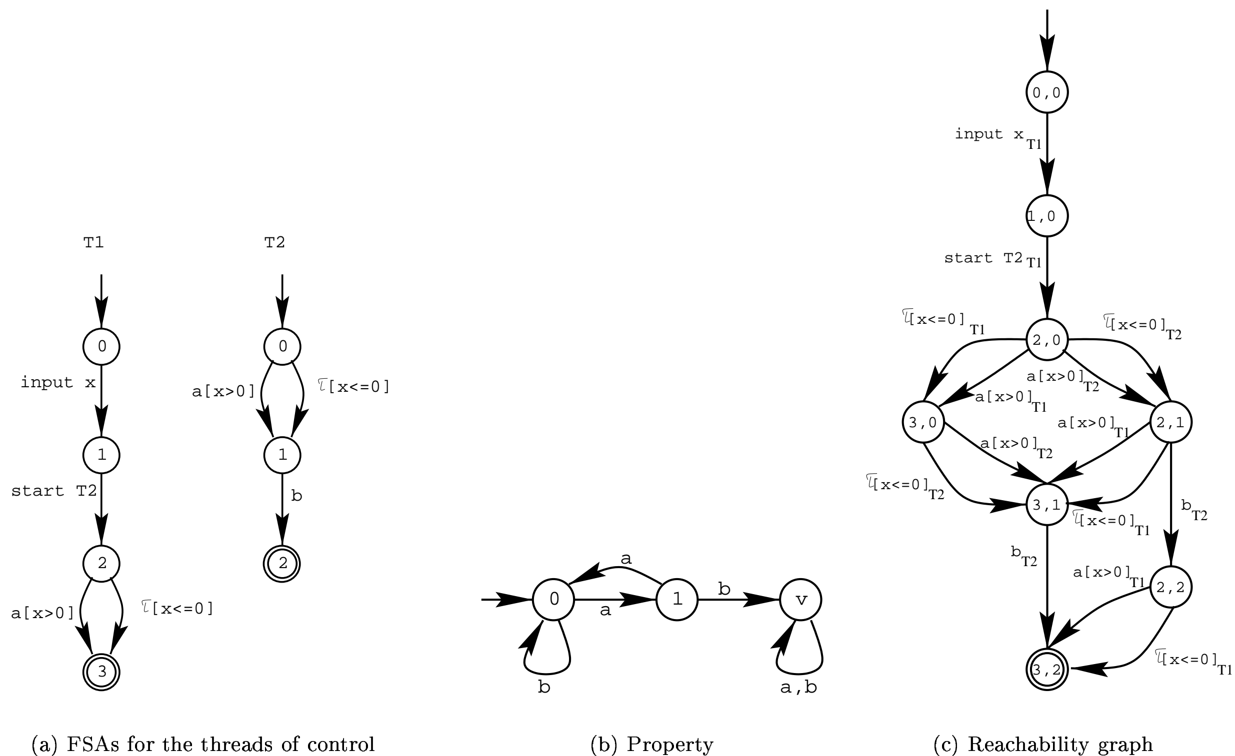
Many FSV approaches are capable of checking two general kinds of properties, safety and liveness. Safety properties are always finitely refutable and liveness properties are never finitely refutable [1]. The approach proposed in this paper deals only with safety properties, since the infinite nature of liveness properties means that an execution that represents a violation of a liveness property is infinite and thus cannot be reasoned about using testing techniques[1].

The goal of our approach is to combine FSV and testing to either prove, with respect to a given property, that no violation states exist in the reachability graph or to find an executable path from the start state to a violation state of the reachability graph.

2.2 Background on Testing

Whereas FSV primarily aims to prove that the specification is satisfied, testing aims to find faults, i.e., to demonstrate that the specification is not satisfied. To test a piece of software, one selects test cases from the

[1] In addition to safety and liveness properties, there are properties that are neither safety nor liveness, but any such property can be represented as a conjunction of a safety property and a liveness property [2]



(a) FSAs for the threads of control

(b) Property

(c) Reachability graph

Figure 1: A reachability graph-based example

input domain, executes the software on each test case, and checks whether the results satisfy the specification. In addition, one might monitor which path through the program is executed by each test case (or other aspects of the execution that are not immediately observable) or might attempt to force execution of a particular path.

Many testing techniques involve analyzing the control flow (and/or data flow) of the program then requiring the test data to execute representatives of certain classes of program paths. These techniques were originally developed for testing sequential programs but can be extended to testing concurrent programs [13, 18, 19]. Testing criteria of this nature often result in a large number of test requirements, even for moderate-sized sequential programs. For concurrent programs, the explosion in the size of the state space makes this problem even more severe. Thus, the tester needs guidelines for selecting portions of the state space that should be explored. In the proposed technique, those guidelines are supplied through interaction with FSV.

One of the most difficult aspects of testing is the *oracle* problem, i.e., the problem of determining whether the result of a particular test case satisfies the specification. The use of formal specifications can significantly alleviate this problem, by allowing test results to be checked automatically [17]. In particular, techniques have been developed for automatically generating test oracles from specifications written in temporal logics, as are commonly used in FSV [7, 8, 16].

In testing and debugging concurrent programs, special problems arise due to non-determinism. A given test case may expose a fault on some executions, but not expose it on others, due to differences in the interleavings of statements from different processes. If executing test case t does not expose a fault, it may be useful to re-execute it many times to check different interleavings. If executing test case t does expose a fault, it may be difficult to reproduce the interleaving in order to debug the program. In order to deal with these issues, testing environments for concurrent programs have been proposed in which the interleaving of processes is monitored or controlled [3].

3 Using Property Violations to Guide Testing

When the finite state verifier finds a property violation, we would like to use this information to guide testing. There are two ways in which we would like to guide testing of concurrent systems, by choosing appropriate test data and by choosing scheduling of relative execution of the threads in cases where they can execute independently from each other. The former is a general problem of testing methods and the latter is specific to concurrent systems. In this section we describe several different approaches to using information produced by reachability analysis to guide testing-based search for property violations.

3.1 Choosing Thread Scheduling

We will assume that our testing approach has instrumentation that lets us at any point to force execution of the current instruction from any of the threads that are not blocked. (Such instrumentation can be either embedded in the run-time execution environment or done on the source code level, similar to [3].) We use information about the violation states in the reachability graph to force testing to exercise those thread interleavings that improve chances of finding a real execution. To this end, we introduce the notion of *interleaving selection criterion ISC* as a predicate defined on the set of all transitions in the reachability graph. This criterion evaluates to true if the transition should be explored, if possible, during testing and false if the preceding run of the verifier does not indicate that taking this transition can lead to a violation state. During testing, we apply this criterion to all transitions that correspond to thread interleavings that can be taken from the current state. If multiple transitions may be taken, according to the criterion, the testing tool will pick one of them and thus drive execution of the test case. If no transition from the current state of the reachability graph can be found that satisfies the criterion, the testing tool will backtrack to an earlier point in the execution and pick an alternative interleaving.

There are two different forms in which verifiers can return information about property violations. One of them is a set of violation states in the reachability graph and the other is a set of paths to some violation states in the reachability graph. Suppose first that V is the set of violation states returned by FSV. An intuitive criterion based on this set is

$$ISC_V(s_1 \xrightarrow{a} s_2) = \begin{cases} \text{false} & \text{if } \forall v \in V, v \text{ is not reachable} \\ & \text{from } s_2 \\ \text{true} & \text{otherwise.} \end{cases}$$

Consider Figure 1(c) and violation state $(2,2)$ found by the verifier used. Suppose that the value of `x` was randomly chosen to be 5 when executing code corresponding to the transition between states $(0,0)$ and $(1,0)$. Consider the point during program execution immediately after thread `T2` has been started by thread `T1`, which corresponds to state $(2,0)$ of the reachability graph. In this state, the two threads may be executed in parallel, and so different event interleavings are possible. One possibility is to execute the `if` statement of `T1`, which means event `a`, because of our choice of value of `x`. This corresponds to the transition to state $(3,0)$ of the reachability graph. Since the violation state $(2,2)$ is not reachable from $(3,0)$, this interleaving will not lead to the violation found by the FSV session, and so will not be taken during testing. The other possible interleaving at state $(2,0)$ is to execute the `if` statement in thread `T2`, which corresponds to the transition on `a` from $(2,0)$ to $(2,1)$. Similarly, out of two possible interleavings at state $(2,1)$, the testing run will choose executing the code corresponding to event `b` in `T2`. At this point, we have detected a violation of the property with testing.

Many verifiers are capable of returning a path or a set of paths to some violation states in the reachability graph. Suppose that W is such a set of paths. An intuitive criterion based on this set is

$$ISC_W(s_1 \xrightarrow{a} s_2) = \begin{cases} \text{false} & \text{if } \forall w \in W, w'(s_1 \xrightarrow{a} s_2) \text{ is not a prefix of } w, \\ & \text{where } w' \text{ is a path traversed up to state } s_1 \\ \text{true} & \text{otherwise} \end{cases}$$

This criterion stipulates that a transition should not be explored if it cannot lead to execution of a path in W. Assume that the verifier returned a violation path $(0,0) \xrightarrow{\texttt{input x}} (1,0) \xrightarrow{\texttt{start T2}} (2,0) \xrightarrow{\texttt{a[x>0]}} (2,1) \xrightarrow{\texttt{b}} (2,2)$. Consider a point of the program execution corresponding to state $(2,0)$ of the reachability graph. If the `if` statement of `T1` is executed at this point, this path will not be followed, and so the testing tool has to execute the `if` statement of `T2`.

Intuitively, ISC_W is stronger than ISC_V in the sense that following a violation path during testing (if it is feasible and test data are adequate) always leads to a violation of the property, while entering a violation state does not necessarily represent a violation, because the path taken to this violation state during testing may be different from any of the paths that represent the violation.

There may be situations in which ISC_W is preferable and situations in which ISC_V is preferable. State $(3,2)$ of the reachability graph is a violation state, since the graph contains the path $(0,0) \xrightarrow{\texttt{input x}} (1,0) \xrightarrow{\texttt{start T2}} (2,0) \xrightarrow{\texttt{a[x>0]}} (2,1) \xrightarrow{\tau\texttt{[x≤0]}} (3,1) \xrightarrow{\texttt{b}} (3,2)$ that violates the property. (There is also another violation path to this violation state.) Suppose first that we use this violation path in the interleaving selection criterion. Since this path is spurious, no choice of test data will exercise it. Thus, on each test case, the testing tool will stop execution because the given path cannot be exercised, even though these test cases could potentially execute a different violation. Now suppose that we use violation state $(3,2)$ in the testing criterion. Even though none of the violation paths to this state are feasible, testing could still find a violation by examining a real violation path $(0,0) \xrightarrow{\texttt{input x}} (1,0) \xrightarrow{\texttt{start T2}} (2,0) \xrightarrow{\texttt{a[x>0]}} (2,1) \xrightarrow{\texttt{b}} (2,2)$. Note that this path leads to a different violation state, $(2,2)$, but ISC_V permits that, because state $(3,2)$ is reachable from $(2,2)$.

Alternatively, suppose that our testing criterion is based on the violation state $(2,2)$. Suppose that the value of `x` used in our test is negative. In this case, path $(0,0) \xrightarrow{\texttt{input x}} (1,0) \xrightarrow{\texttt{start T2}} (2,0) \xrightarrow{\tau\texttt{[x≤0]}} (2,1) \xrightarrow{\texttt{b}} (2,2)$ can be taken. Even though the violation state is reached, testing did not find a violation of the property, because at the end of this path the property is in state 1, which is not a violation state. If, instead of a violation state, the verifier returns the path $(0,0) \xrightarrow{\texttt{input x}}$

Figure 2: The reachability graph modeling variable x

Figure 3: The proposed process of simultaneous use of FSV and testing

$(1,0) \xrightarrow{\texttt{start T2}} (2,0) \xrightarrow{\texttt{a[x>0]}} (2,1) \xrightarrow{\texttt{b}} (2,2)$, testing with a negative value of x will be stopped early, at state $(2,0)$, because transition $(2,0) \xrightarrow{\texttt{a[x>0]}} (2,1)$ cannot be taken.

3.2 Choosing Test Data

Consider the problem of choosing appropriate test data. In general, choosing input data to follow a path through the reachability graph in such a way that it correlates with the values of modeled variables is undecidable. In practice, it may be possible to use symbolic execution [5] or some heuristics [10, 14] or to use random test data, aborting those executions that are not exploring the part of the reachability graph of that is of interest. In this section we propose an approach for choosing values of variables that are modeled by the verifier.

Many FSV approaches are capable of modeling system variables and including them in the analysis. In this case information about these variables (either the actual values or approximations, such as sets or intervals of values) can be used by testing to choose input data for this variable. For example, behaviors of variable x in our example in Figure 1 can be modeled by including the sign of x in the states of the reachability graph, as shown in Figure 2. Each state in this reachability graph is labeled (s_1, s_2, r), where s_1 is the state of thread T1 from Figure 1(a), s_2 is the state of thread T2, and r is the range of values of variable x. In this example we consider only three possible ranges, $\texttt{x} > 0$, $\texttt{x} \leq 0$, and all, which denotes all possible values of x (the latter appears only in the start state of the reachability graph).

Suppose that we use violation state $(2, 2, \texttt{x} > 0)$ found by FSV to drive testing. The simplest approach in this case is to use the range $\texttt{x} > 0$ in test data selection, since we can perform an additional static analysis and detect that, once selected, the value of x does not change in this program.[2] This choice is even easier if path-based test criterion is used, because it is only the range of values of x that appears in the states along this path that has to be analyzed.

Modeling variables in this way in the reachability graph can help not only in selecting test data, but also in quickly discarding test data that are not likely to lead to violation. For example, if a negative value of x is chosen for testing, then after transition $(0, 0, \texttt{all}) \xrightarrow{\texttt{input x} \leq 0} (1, 0, \texttt{x} \leq 0)$ in the reachability graph in Figure 2 is taken during testing, we can stop execution of the test, because none of the violation states in the reachability graph are reachable from state $(1, 0, \texttt{x} \leq 0)$.

4 Conclusion

We have proposed a technique for using testing and finite state verification synergistically in the attempt to verify or disprove properties of concurrent systems. The process for applying this technique is illustrated in Figure 3. In our approach, testing and model refinement are both used to explore violations returned by the verifier, in order to determine whether those violations represent real executions or are spurious. Information returned by the verifier describing violation states and/or paths to those states is used to guide testing. Test cases are run in an environment where interleavings can be controlled and where executions can be aborted if it is determined that they will not be able to reach given violation states or execute a given paths to violation states.

[2] If x is redefined, the problem can be much more difficult.

This approach may offer a more efficient way to determine whether a violation is spurious than model refinement alone. In addition, finding test data that causes a property violation can be useful for identifying and removing the fault. We plan to implement our approach and to carry out experiments aimed at determining whether it is indeed useful.

References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, Jan. 1993.

[2] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[3] R. Carver and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Software*, pages 66–74, Mar. 1991.

[4] S. C. Cheung and J. Kramer. Compositional reachability analysis of finite-state distributed systems with user-specified constraints. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 140–151, Oct. 1995.

[5] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3), Sept. 1976.

[6] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[7] L. Dillon and Y. Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *ACM SIGSOFT Foundations of Software Engineering*. ACM Press, Oct. 1996.

[8] L. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. In *ACM SIGSOFT Foundations of Software Engineering*, pages 140–153. ACM Press, Dec. 1994.

[9] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 62–75, Dec. 1994.

[10] N. Gupta, A. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *Proceedings Foundations of Software Engineering*. ACM Press, Nov. 1998.

[11] G. Holzmann. Designing executable abstractions. In *Proceedings of the 2nd Workshop on Formal Methods in Software Practice*, pages 103–108, Mar. 1998.

[12] G. J. Holzmann. The model checking SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[13] P. Koppol and K.-C. Tai. An incremental approach to structural testing of concurrent systems. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pages 14–23. ACM Press, Jan. 1996.

[14] B. Korel. Automated software test generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, Aug. 1990.

[15] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[16] T. O'Malley, D. Richardson, and L. Dillon. Efficient specification-based test oracles. In *Second California Software Symposium*, Apr. 1996.

[17] D. J. Richardson. Testing with analysis and oracle support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*. ACM Press, Aug. 1994.

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

[19] S. N. Weiss. A formal framework for the study of concurrent program testing. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 106 – 113. IEEE Computer Society Press, July 1988.