Automated Software Testing Using Model-Checking

John Callahan
Francis Schneider
Steve Easterbrook

NASA Software Research Laboratory
West Virginia University

Abstract

White-box testing allows developers to determine whether or not a program is partially consistent with its specified behavior and design through the examination of intermediate values of variables during program execution. These intermediate values are often recorded as an execution trace produced by monitoring code inserted into the program. After program execution, the values in an execution trace are compared to values predicted by the specified behavior and design. Inconsistencies between predicted and actual values can lead to the discovery of errors in the specification and its implementation. This paper describes an approach to (1) verify the execution traces created by monitoring statements during white-box testing using a model checker as a semantic tableau; (2) organize multiple execution traces into distinct equivalence partitions based on requirements specifications written in linear temporal logic (LTL); and (3) use the counter-example generation mechanisms found in most model-checker tools to generate new test cases for unpopulated equivalence partitions.

1. Introduction

Software developers often use “models” to reason about the design of their systems, but keeping the models and source code in fidelity during development is a difficult task [1]. Typically, a model provides an abstraction for specifying, communicating, and understanding aspects of the expected behavior of a software system. Examples of models include finite state machines [2], functions [3], flow diagrams, process algebras [4], petri nets, and many other formal and informal notations. During development, the code must not only implement behaviors as specified by a model, but a model itself may need to change based on discovered limitations of the implementation environment [5]. Maintaining fidelity between the code and models is important as the software evolves because any divergence may lead to future problems including design errors, inconsistent documentation, and expensive rework.

While it is possible in some cases to generate code directly from a model, most designers must develop software directly in a standard programming language. To ensure that their code reflects their model, developers frequently test their software during development and refine their designs and code based on the results of these tests. Such a process is similar to software prototyping but is done primarily to confirm the viability of implementing a proposed design in a target environment. The use of testing in such circumstances establishes an informal relationship between the code and a design model. Few explicit software development methods, however, exist that support this process of refinement and co-evolution of designs and their implementations. As a result, the behaviors expressed by models and code often diverge later in the development lifecycle because fidelity between them is difficult to maintain as changes are made to either representation.

White-box testing allows developers to establish some fidelity between models of their software designs and their code during development. The output traces of white-box tests, achieved via the use of a debugger or embedded print statements, help to validate that the code behaves in accordance with a model of its design. If an inconsistency between a trace and the specified behavior is discovered, then the model or the code can be corrected as appropriate. For example, Bentley describes the use of monitoring statements in the implementation of a binary search program [6]. In one test case, the input and output of the program are correct, but inconsistencies between the intermediate values of the upper and lower indexes in the execution trace lead him to discover the error in the code. This form of debugging is common and informal, but relies on the existence of an external model of the program's design to compare against the actual behavior. A test oracle (in this case Bentley himself) relies on an intuitive model of binary search to verify that the program behaves correctly during execution.

Specification-based testing advocates have long promoted the use of formal models as a source for test generation and test oracles [7, 8]. Our approach, called *formal testing*, is a specification-based testing process that uses model checking techniques to verify, organize, and generate white-box tests during evolutionary software development. While a model can be analyzed directly using model checking methods for safety, invariance, liveness, and other properties, it can also be used to manage and organize a test environment. We have developed a technique to verify execution traces during white-box testing using a model-checker as a semantic tableau [9]. We also show how requirements stated as linear temporal logic formulae can be used to organize execution traces into equivalence partitions [10] to determine the adequacy of test coverage relative to a set of requirements. Finally, the use of model checking allows for automated generation of tests for unpopulated partitions. This approach identifies inconsistencies between the code and models and helps to leverage more powerful forms of analysis throughout the development process.

2. Model Checking

A model checker takes a description of several concurrent, finite state machines as input and effectively analyzes the expanded computation tree for given properties. A *computation tree* is a conceptual structure that consists of a possibly infinite set of all possible execution paths. For example, consider the concurrent machines *P1* and *P2* which cycle through a sequence of states 1→2→3→1→…. When either process reaches state 3, it will reset the state of both machines to state 1. Figure 1 depicts the computation tree consisting of all possible execution paths in the joint machine $P1 \cup P2$ where each node is denoted by the composite state of each subprocess.

The computation tree can be searched effectively to ensure that all paths obey specific constraints expressed as liveness, invariance, and safety properties. Model checkers employ various methods to reduce the complexity of the search. For example, redundant states can be eliminated from searches due to the memory-less properties of finite state machines. Redundant exploration of the subtree below the joint state (2,2) in Figure 1 can be eliminated from consideration. Other related optimization techniques include partial order reduction [11] and the use of binary decision diagrams (BDDs) in symbolic model checkers [12].

Through the effective expansion of the computation tree, the behavior of the model can be analyzed for specific properties. Such properties can be specified as linear temporal logic (LTL) formulae that describe sets of paths (possibly empty) in the computation tree. LTL formula employ many temporal operators [13] including the always ([]), next (O), and eventually (<>) operators. For example, the LTL formula [](p1=1) describes the starvation of process 1. Given this formula, a model checker would identify a single path in the computation tree that corresponds to the rightmost, infinite path in Figure 1. Such formulae can

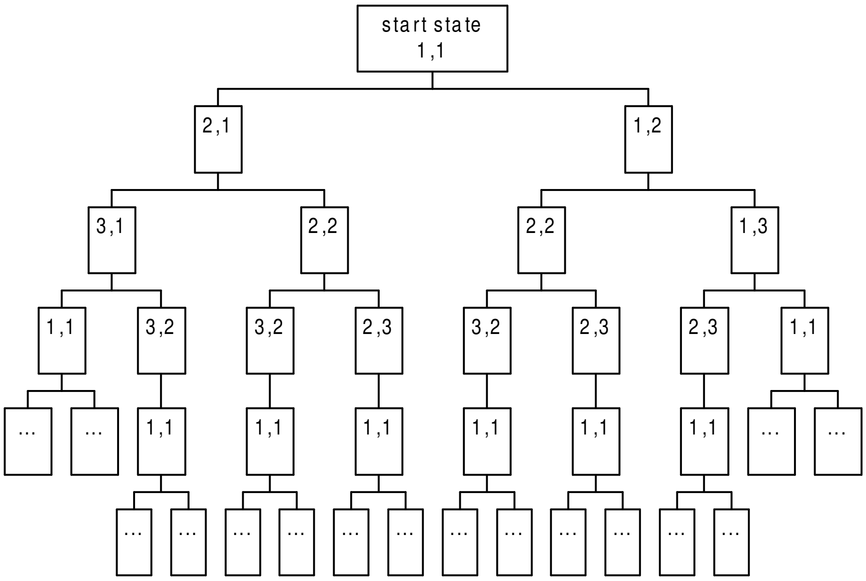

Figure 1: infinite computation tree for concurrent finite-state machines *P1* and *P2*

describe non-trivial patterns for multiple paths that have different behaviors but satisfy a common property. For instance, starvation of either process corresponds to the leftmost and rightmost paths in Figure 1.

3. Requirements as Temporal Formulae

A finite state model can specify a design solution for meeting a set of requirements. Required behaviors of the system can be expressed as temporal constraints on a design model. A model checker can be used to determine whether or not the model contains paths that satisfy a specific property. There are three categories of properties that correspond to set of paths in a model:

- no paths in the model should exhibit the property (i.e., a safety property)
- all paths in the model should exhibit the property (i.e., an invariance property)
- some paths in the model should exhibit the property (i.e., a liveness property)

For example, in the case of the machine $P1 \cup P2$ shown in Figure 1, we can specify a requirement R_1 such that in some cases *P1* or *P2* must completely execute to state 3 before letting the other process execute (i.e., the execution of *P1* and *P2* is *not* interleaved). This corresponds to a liveness property satisfied in the model by some of the paths in the computation tree shown in Figure 1. The complement of a property is expressed by its negation. The union of paths that satisfy a liveness property and its negation include all paths in a model.

Safety requirements are those properties which *none* of the paths in a model satisfy. To check a model for the absence on all paths of specific behavior means that effectively all paths in the model have to be explored. This corresponds to our intuitive notion that in order to check for the absence of a property, then exhaustive testing of all paths for a safety property is necessary but often infeasible. An invariant property is one that *all* paths in the model must satisfy. An invariant property is the logical complement of a safety property. For example, we can require that the system must always eventually reach the state of (1,1) in the computation tree in Figure 1. This property must be true of all paths in a computation tree. Like safety properties, invariant properties require an effective search of all paths in the model to determine its presence.

For our example, we can express some requirements as temporal formulae:

$$[]<>([1,3] \vee [3,1]) \qquad [R_1]$$

$$[]<>([2,1]) \qquad [R_2]$$

$$[]<>([3,2]) \qquad [R_3]$$

The requirement R_1 specifies that the execution of processes *P1* and *P2* must not be interleaved. It states that *always eventually* the joint state [1,3] or [3,1] must appear on the execution path. This holds for any path on which that state [1,3] or [3,1] is reachable from all the nodes on that path. Its complement property, ~R_1, asserts that neither joint state [1,3] nor [3,1] is reachable. Another property, R_2, states that process *P1* must be the first process to execute some of the time. Its complement property, ~R_2, specifies that process *P2* must be the first process to execute some of the time. A final property, R_3, specifies that process *P1* must terminate last under interleaved execution and its complement ~R_3 specifies that process *P2* must terminate last.

Each of these properties identify a set of infinite paths in the computation tree. For validating, partitioning, and generating test sequences, we need only consider *finite prefixes* of these paths that exhibit the desired properties. Otherwise, each finite path as shown in Figure 1 is a prefix of a path that exhibits all the properties listed above. Fortunately, the counter-example mechanism in most model checkers produce such prefixes (including fixed cycles) so that we can reason about properties of finite test traces.

4. Testing based on Model Checking

We use the SPIN model checker [14] and its counter-example generation mechanism to validate test traces, organize tests into equivalence partitions, and generate new tests for unpopulated partitions. We assume that a partial, finite-state model of the system exists and that system requirements can be stated as temporal properties of the model. A SPIN model is specified in the Promela language as a finite set of asynchronous, concurrent processes that interact through shared variables and communication channels (a special case of shared variable).

Given the Promela specification, the SPIN model checker explores the computation tree for the presence of paths that exhibit a given property. The property is specified as a special, synchronous process called a never claim. The never claim is implemented as a Bucchi automata that terminates when a property is exhibited by a path in the computation tree. If a never claim succeeded (i.e., the Bucchi automata terminates or enters an accepting state), the SPIN model checker will produce a trace path as a counter-example that exhibits the property in question and the user can resume the search for the next path or terminate the search.

4.1 Trace Validation

In this first use of model-based testing, we do not employ the Bucchi automata. We use white-box testing to create execution traces of a program's behavior and use the SPIN model checker as a semantic tableau to determine whether or not a trace corresponds to a path within the computation tree of a model. First, the program under test is instrumented with statements that produce an execution trace of the intermediate values of important program variables. This allows us to monitor the changes of state within the program during and after execution to determine if program's behavior concurs with the design specified by the finite state model. Next, we validate an execution trace by creating an additional trace process (P_T) within the model that contains the event sequence specified by the trace. The process P_T proceeds based on events and variable values within other processes in the model. A false assertion is appended to the end of the trace to force the generation of a counter-example if the process P_T terminates. The process P_T can only terminate for legal paths in the computation tree created by other processes in the model. If P_T terminates and produces a counter-example, then the test trace concurs with the behavior specified by the model.

Failure to generate a counter-example means that the trace is not in the computation tree specified by the finite-state model. This does not imply, however, that the program is incorrect or the model is incorrect. Inconsistencies may be due to the lack of sufficient instrumentation, incompleteness of the model or errors in either the model or code or both. Such inconsistencies, however, help to maintain a higher degree of fidelity between the model and code. They also serve to guide placement of instrumentation in the code.

Likewise, acceptance by the model (i.e., generation of a counter-example) is not definitive. If a trace contains only a subset of intermediate states then the model could recognize the trace in the model, but the behavior may not account for all variable changes. Consider the trace

$$[1,1] \rightarrow [2,3] \rightarrow [1,1]$$

that corresponds to a subset of the states in the computation tree in Figure 1. While the trace is valid using our approach, it accounts for only a subset of nodes along some paths in the computation tree. To solve this problem requires the introduction of a counter in the model to synchronize events in the model and the traces, but such a counter exacerbates the state space explosion problem in the model checker. This requires that the program be instrumented at all points where the variables of interest are modified. With built-in "watchers" in most debuggers, this is usually not a problem.

4.2 Equivalence Partitioning

Test analysts use equivalence partitioning on the input space of a program to minimize the number of tests needed to cover all expected behaviors of a software system under test [10]. Creating partitions, however, is usually an informal and difficult task. Traditionally, the input space is partitioned roughly into overlapping subsets based on some categorization of the expected output behavior of the program under test. In white-box testing, the expected behavior includes the intermediate values of internal variables as shown in the execution trace during the test.

Our approach involves partitioning paths in a computation tree based on combinations of requirements and their logical complements. We construct a partitioning based the conjunction of all requirements and their complements as shown in Table 1. This partitioning is called the *conjunctive complementary closure* (CCC) of a set of requirements. Each combination is called a *coverage property* because it describes a

Partition number	Conjunctive complementary closure (CCC)		
1	R1	R2	R3
2	R1	R2	~R3
3	R1	~R2	R3
4	R1	~R2	~R3
5	~R1	R2	R3
6	~R1	R2	~R3
7	~R1	~R2	R3
8	~R1	~R2	~R3

Table 1: Equivalence partitions on R1, R2, and R3

unique set of paths. While the partitions created by a CCC are disjoint[1], this only applies to complete paths in the computation tree. Path prefixes may fall into one or more partitions.

Since paths in a computation tree are usually infinite, we cannot enumerate paths in the CCC partitions of most computation trees, but we can "sort" execution traces for specific tests into one or more partitions using a model checker. We can determine if a trace belongs to a partition by (1) adding the a trace process P_T to a model as we did for trace validation and (2) adding a synchronous, Bucchi automata that enters an accepting state if a specific coverage property is satisfied. Using a special feature of the SPIN model checker, we can automatically generate Bucchi automata in Promela from linear temporal logic (LTL) specifications. We then append the generated Bucchi automata to the end of the model. The Bucchi automata must be in an accepting or terminal state and the trace process P_T must terminate in order to generate a counter-example. Instead of terminating with a false assertion as in the case of trace validation, the trace process sets the variable `done` to the value 1. If the trace process terminates and the Bucchi automata terminates or is in an accepting state, then SPIN will generate a counter-example and the trace belongs in the partition specified by the LTL formulae.

A test trace may fall into one or more partitions. Indeed, test traces of sufficient length can fall into all partitions of the CCC. A test trace that falls into one and only one partition is called *a minimal test trace*. For N coverage properties, a set of N minimal tests represents the minimum test suite needed to exercise the program relative to the CCC. Such minimal test traces can usually be generated and consist of the shortest prefixes that exhibit a given coverage property.

Some coverage properties will not be found in a computation tree for a model. For example, the requirements in partition number 1 are inconsistent because R1 requires non-interleaved behavior while R3 requires interleaved execution. While this can be detected by inspection, the model checker will detect this inconsistency automatically by (1) failing to populate that partition with any valid test traces and (2) failing to generate valid test traces (see section 4.3 below).

A coverage property that corresponds to a non-empty partition is called a *valid coverage property*. Invalid coverage properties are the result of conflicts between requirements. It is expected that no test traces

---

[1] The proof that CCC partitions are disjoint is by contradiction: assume that a path P satisfies at least two coverage properties C1 and C2. But C1 and C2 will differ on at least one requirement and its negation by definition of the CCC set. A path cannot satisfy a requirement and its negation. Therefore, the assumption is false and CCC partitions must describe disjoint sets of paths in the model.

should fall into such partitions. Paths that fall into invalid partitions are interesting because they can be used to identify errant behavior in the implementation and to generate tests that stress the implementation for cases that should not occur. So rather than eliminate such partitions from the model, we use them to identify problems and generate stress tests.

4.3 Test Generation

Most model checkers include mechanisms for producing counter-examples if some paths in a computation tree exhibit a given property. The SPIN [14], Murphi [15], and SMV model checkers [16], for example, will produce counter-examples when paths exist in the computation tree that violate assertions or satisfy temporal formulae. We can cause the SPIN model checker to generate counter-examples on demand for a given coverage property. As in partitioning, we use the Bucchi automata mechanism in SPIN to analyze the behavior of the state model, but we do not use an additional trace process. If a coverage property is valid, the SPIN model checker produces a set of trail files that correspond to paths that exhibit the failed behavior.

The SPIN model checker will produce finite prefixes or fixed cycles that exhibit a given property if it exists in the computation tree. These counter examples serve as *test templates* for constructing actual test input sequences on an implementation. Partitions that correspond to invalid coverage properties can be used to detect inconsistencies in the requirements themselves and to generate off-nominal test cases that can be used to stress test an implementation (i.e., such template traces can be used to try and force the implementation into failure modes).

5. Scalability

As another example, consider a railroad crossing scenario consisting of 3 separate processes: a train, a gate, and a car. Two obvious properties of this system are that

1. The car and train should never cross at the same time

2. The car should eventually get to cross (i.e., the gate does not stay down indefinitely)

The first property is a safety condition while the second is a liveness condition. We first construct a naïve model consisting of the 3 processes and synchronization between changes in their states. For example, the gate would transition from the UP position to DOWN when the train is in state APPROACHING.

However, analysis of the naïve model using a model checker would quickly identify violations of both of these properties. In the first case, a race condition exists between the train and the gate, i.e., the train may "beat" the gate by reaching the crossing before the gate is down thus allowing the car to cross while the train is at the intersection. In the second case, without a sense of fairness in the model checker, the car could be caused to wait indefinitely on an infinite number of trains.

We claim that the presence of these paths in the computation tree of a naïve model of the railroad system can be useful in several ways. First, if a test trace is classified in one of the partitions corresponding to such a property, it is clearly a failure case of the implementation. Second, we can generate test templates for these errant partitions to stress test the implementation for such off-nominal behaviors.

We have applied this technique in an informal manner to a complex Internet protocol for reliable multicasting [17]. Formal testing allowed us to keep the protocol state model in agreement with the code during the implementation phase of the protocol engine. Due to implementation considerations caused by memory limits, network performance and the changing needs of applications using the protocol, some aspects of the protocol requirements had to be changed during implementation. Formal testing allowed us to keep the model and code in synch with each other, organize our test cases, remove obsolete tests cases, generate new test cases for unseen conditions, and perform analysis for invariant and safety properties on the protocol throughout the entire development lifecycle.

6. Future Work

Future work includes exploration of the sources of conflicts between requirement formulae that lead to invalid coverage properties. Some of the conflicts may arise due to inconsistencies in the requirements themselves. We may be able to exploit conflicts to eliminate non-viable test partitions or reduce partitions to partial combinations of requirements that interact in a non-trivial manner. Exploration of the sources of conflicts between requirements may also lead to the discovery of incomplete behaviors in the model.

We are also exploring the dynamics of software development processes that employ formal testing as a means to evolve software and specifications simultaneously. We believe that although the ideal situation is to analyze a set of requirements fully before implementing them, the reality of software development is that specifications and implementations must change during the entire lifecycle of a software system even after deployment. The changes to either specification or code must be synchronized with the other.

Furthermore, developers must be able to assess the impact of changes in either specifications or code on the other. For example, by using formal testing, we can assess the impact of a specification change in terms of the percentage of existing tests that are invalidated (i.e., are no longer traces in a valid equivalence partition) or reclassified (i.e., a member of different equivalence partition) due to the change. Current test management techniques are inadequate for determining the regression impact of specification changes on test suites.

7. Summary

Testing remains a powerful and intuitive approach to ensuring the quality and reliability of software, but testing also has serious limitations. We believe that by managing testing via a formal methods we can reap the benefits of formal analysis on a model that is kept in fidelity with the code. The model can be analyzed for invariant and safety properties that are difficult to test for completely. The composition of both testing and formal methods in this manner has great potential since both approaches complement the strengths and weaknesses of the other method.

8. References

1. Murphy, G.C., D. Notkin, and K. Sullivan, *Software Reflexion Models: Bridging the Gap between Source and High-Level Models*, in *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium*. 1995. p. 18-28.

2. Heitmeyer, C., B. Labaw, and D. Kiskis. *Consistency checking of SCR-style Requirements Specifications*. in *Second IEEE International Symposium on Requirements Engineering*. 1995.

3. Stocks, P. and D. Carrington, *A Framework for Specification-based Testing*. IEEE Transactions on Software Engineering, 1996. **22**(11): p. 777-793.

4. Jackson, D. and E.J. Rollins, *A New Model of Program Dependences for Reverse Engineering*, in *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations*. 1994. p. 2-10.

5. Swartout, W. and R. Balzer, *On the Inevitable Intertwining of Specification and Implementation.* j-CACM, 1982. **25**(??): p. 438-440.

6. Bentley, J., *More Programming Pearls*. 1988, New York: Addison-Wesley.

7. Howden, W., *The Theory and Practice of Functional Testing.* IEEE Software, 1985. **2**(5): p. 6-17.

8. Poston, R.M., *Automated testing from object models.* j-CACM, 1994. **37**(9): p. 48-58.

9. Dillon, L.K. and Q. Yu, *Oracles for Checking Temporal Properties of Concurrent Systems*, in *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations*. 1994. p. 140-153.

10. Myers, G., *The Art of Software Testing*. 1979, New York: John-Wiley.

11. Peled, D. *Combining partial order reductions with on-the-fly model checking*. in *6th International Conference on Computer Aided Verification*. 1994. Stanford, CA.

12. Clarke, E. and E. Emerson. *Characterizing Properties of Parallel Programs as Fixed Points*. in *7th International Colloquia on Automata, Languages and Programming*. 1981: LNCS 81.

13. Manna, Z. and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. 1992: Springer-Verlag.

14. Holzmann, G., *Design and Verification of Computer Protocols*. 1991: Prentice-Hall.

15. Dill, D., *et al. Protocol Verification as a Hardware Design Aid*. in *IEEE Conference on Computer Design: VLSI in Computers and Processors*. 1992: IEEE Computer Society Press.

16. Burch, J., *et al.*, *Symbolic Model Checking for Sequential Circuit Verification.* IEEE Transactions on Computer-Aided Design, 1994. **13**(4).

17. Callahan, J. and T. Montgomery. *Approaches to Verification and Validation of a Reliable Multicast Protocol*. in *Proceedings of the International Symposium on Software Testing and Analysis*. 1996. San Diego, Ca: Association for Computing Machinery.