Specification Centered Testing

Mats P. E. Heimdahl
University of Minnesota
4-192 EE/CS Building
Minneapolis, Minnesota 55455
USA
heimdahl@cs.umn.edu

Sanjai Rayadurgam
University of Minnesota
4-192 EE/CS Building
Minneapolis, Minnesota 55455
USA
rsanjai@cs.umn.edu

Willem Visser
NASA Ames Research Center
M/S 269-2
Moffett Field, CA 94035-1000
USA
wvisser@ptolemy.arc.nasa.gov

ABSTRACT

This position paper discusses a framework for automating the testing of systems with stringent structural coverage requirements, for example, avionics systems. The framework covers testing of the model of the desired behavior as well as the resulting implementation. We use a formal model of the required software behavior as the central component of our testing strategy; we call this approach *specification centered testing*. We discuss how a model checker can be used to automatically generate complete test sequences that will provide arbitrary structural coverage of the requirements specification as well as the code implementing the requirements.

1. INTRODUCTION

Software development for critical embedded control systems, such as the software controlling aeronautics applications and medical devices, is a costly and time consuming process. In such projects, the validation and verification phase (V&V) can consume 50%–70% of the software development resources. Although there have been breakthroughs in static V&V techniques, such as model checking and theorem proving, *testing* is still an invaluable V&V technique that *cannot* be replaced. Currently, the majority of the V&V time is devoted to the development of test cases to adequately test the required functionality of the software (black-box, requirements-based testing) as well as adequately cover the implementation (white-box, code-based testing). Thus, if the process of deriving test cases for V&V could be automated and provide requirements-based and code-based test suites that satisfy the most stringent standards (such as, for example, DO-178B–the standard governing the development of flight-critical software for civil aviation), dramatic time and cost savings would be realized.

In this paper, we present a *specification-centered* approach to testing where we rely on a formal model of the required software behavior for test-case generation, as well as, an oracle to determine if the implementation produced the correct output during testing. We also discuss how the specification-based tests can be augmented with tests derived from the code. Our work is based on the hypothesis that *model checkers* can be effectively used to automatically generate test sequences that provide a predefined *structural coverage* of a formal specification. In [15], we defined a formalism suitable for representing software engineering artifacts in which various structural test coverage criteria can be defined. In [14] we demonstrated how this formal foundation is used to generate structural tests from a formal specification of the required software behavior. In this work, we used RSML^{-e} as the specification language [18, 20]. Here we will discuss our overall framework and discuss some research challenges.

The rest of the paper is organized as follows. Section 2 provides a short overview of related efforts. We then discuss our overall approach in Section 3. Finally, we provide a short discussion of our work and the conclusions.

2 BACKGROUND AND RELATED WORK

Much research has gone into both model checking and software testing. The coverage in this section is by necessity cursory, but we will present the research efforts that are most relevant to our current work.

2.1 Testing and Model Checking

Model checking is a form of formal verification where a finite state representation of a proposed system can be explored exhaustively to determine if it satisfies desirable properties. The properties to be verified are cast as assertions of formulas in an appropriate temporal logic and the

system behavior is specified as some form of a transition system. Temporal logic formulas typically make claims about system behavior as it evolves over time. Both temporal logics and model-checking have been active research areas for more than a decade [7, 5]. Many popular temporal logics like Computation Tree Logic (CTL and CTL*) [7] and Linear-time Temporal Logic (LTL) [13] have associated model-checking systems such as SPIN [10], SMV [11], and new extensions to PVS [12, 16].

Model checkers build a finite state transition system and exhaustively explore the reachable state space searching for violations of the properties under investigation. Should a property violation be detected, the model checker will produce a counter-example illustrating how this violation can take place. In short, a counter-example is a sequence of inputs that will take the finite state model from its initial state to a state where the violation occurs. We use the model checker as a test data generator by posing various test coverage criteria as challenges to the model checker. For example, we can assert to the model checker that a certain transition in a specification cannot be taken. If this transition in fact can be taken, the model checker will generate a sequence of inputs (with associated outputs) that forces the model to take this transition—we have found a test case that provides transition coverage of this transition. With this approach we will be able to automatically generate test suites for user defined levels of specification test coverage.

2.2 Test Data Generation

Although much work has gone into definition and evaluation of various test selection strategies [21], the ability to automate the selection of tests is limited.

To our knowledge, two other research groups are pursuing the promising approach to use model checking to generate test cases. Gargantini and Heitmeyer [8] describe a method for generating test sequences from requirements specified in the SCR notation. To derive a test sequence, a *trap property* is defined which violates some known property of the specification. In their work, they define trap properties that exercise each case in the event and condition tables available in SCR—this provides a notion of branch coverage of an SCR specification. A model-checker is then used to produce a counter-example to the trap property. The counter-example generated assigns a sequence of values to the abstract inputs and outputs of the system, thus making it a test sequence. Our work differs in primarily two ways. First, our focus is on test sequence generation for a rich collection of *structural coverage criteria* (both path-based and condition-based) and our goal is to develop techniques to produce test sequences that provide desired coverage without being excessively large. Second, we do not tie the formal definitions of coverage nor the test sequence generation to any particular formalism (such as SCR), instead we base our criteria and generation on an underlying formalism. This makes our results language independent and widely applicable to any notation that can be model checked (including SCR).

Ammann and Black [2, 1] combine mutation analysis with model-checking based test case generation. They define a specification based coverage metric for test suites using the ratio of the number of mutants killed by the test suite to the total number of mutants. Their test generation approach uses a model-checker to generate mutation adequate test suites. The mutants are produced by systematically applying mutation operators to both the properties specifications and the operational specification, producing respectively, both positive test cases which a correct implementation should pass, and negative test cases which a correct implementation should fail. Their work focuses solely on the specification and on mutation analysis. Our proposed approach takes a broader view and includes more traditional test selection criteria that are based on the structure of the software artifact being tested.

Blackburn and Busser [3] use a different method for test case generation that does not use a model checker. In this approach, each test vector is a pre/post pair of system states. Also, Simon Burton at the University of York [4] discusses the use of theorem-proving approach for generating test cases. However, both approaches do not generate a sequence of inputs that leads to the state pair from the initial state of the system.

3 TEST GENERATION FRAMEWORK

General Testing Framework: Figure 1 shows an overview of our proposed *specification-centered testing* approach. During the development of a formal requirements model, the model must be extensively inspected and analyzed, as well as extensively tested and simulated (step 1 in Figure 1)—an approach we advocate in *specification-based prototyping* [18, 17].

A set of tests are developed from the informal requirements to evaluate the required functionality of the model (functional tests). These functional tests may not, however, reach all the parts of the formal model—there may be transitions and conditions not exercised by these tests. The functional tests will in most cases have to be complemented by a collection of white-box tests developed specifically to exercise a specification up to a certain level of *specification coverage*. This test case generation is part of our current efforts and has been described in [15, 14]. We use a model checker to automatically derive test suites providing user specified coverage of the specification (specification-based structural tests).

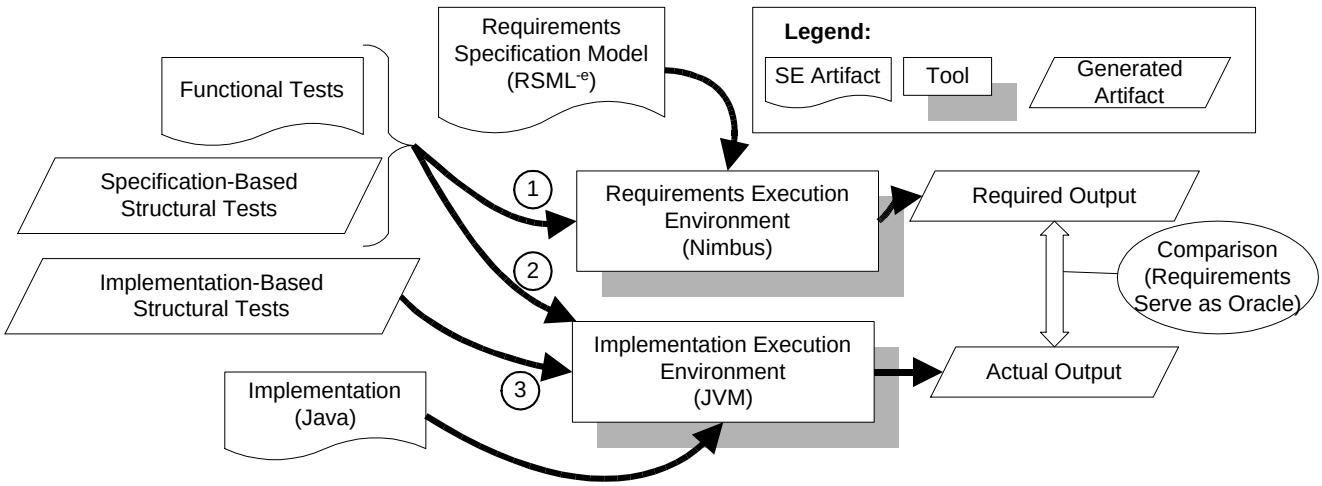

Figure 1. An overview of the specification centered testing approach.

After we have generated test sequences from the specification, for instance, to provide basic condition coverage of the specification, all tests used during the testing of the formal specification can naturally be reused when testing the implementation in a later stage (step 2 in Figure 1). The test cases derived to test the formal specification provide the foundation for the testing of the implementation and the executable formal specification serves as an oracle during the testing of the implementation; thus, *specification centered testing*.

Finally, the specification based test set may not provide adequate coverage of the implementation—it will most likely have to be augmented with additional test cases (step 3 in Figure 1). With the recent advances in code model checking, for example, JPF [19], VeriSoft [9], and the Bandera tools [6] we believe the use of model checkers is also applicable when deriving test cases based on the implementation.

Finding Tests with a Model Checker: The general approach of finding test sequences using model checking techniques is outlined in Figure 2.

A model checker can be used to find test cases by formulating a test criterion as a verification condition for the model checker. For example, we may want to test a transition (guarded with condition C) between states A and B in the formal model. We can formulate a condition describing a test case testing this transition—the sequence of inputs must take the model to state A; in state A, C must be true, and the next state must be B. This is a property expressible in the logics used in common model checkers, for example, the logic CTL. We can now challenge the model checker to find a way of getting to such a state by negating the property (saying that we assert that there is no such input sequence) and start verification. The model checker will now search for a counterexample demonstrating that this property is, in fact, satisfiable; such a counterexample constitutes a test case that will exercise the transition of interest. By repeating this process for each transition in the formal model, we use the model checker to automatically derive test sequences that will give us transition coverage of the model. The proposed test generation process is outlined in Figure 2.

Our approach to defining test criteria is based on the structure of the specifications, analogous to structural coverage criteria defined for code. The goal is to ensure that test cases cover various constructs in the specification. Thus the criteria themselves are language dependent while the formal framework in which they are expressed could be applied to any similar specification language or code. We have formalized a transition system suitable for modeling of common specification languages, such as RSML^{-e}, SpecTRM-RL, and SCR, or code. We have demonstrated how various structural coverage criteria can be defined in terms of this framework [15]. Examples of coverage criteria for which we can derive test cases include state coverage, condition coverage, and MC/DC coverage. We demonstrated the application of this approach on RSML^{-e} in [14].

4 DISCUSSION AND CONCLUSION

We have outlined a framework designed to help automate test-case generation for software engineering artifacts such as code and formal specifications. We use structural coverage criteria to define what test cases are needed and the test cases are then generated using the power of a model checker to generate counter-examples. Our approach is to cast the

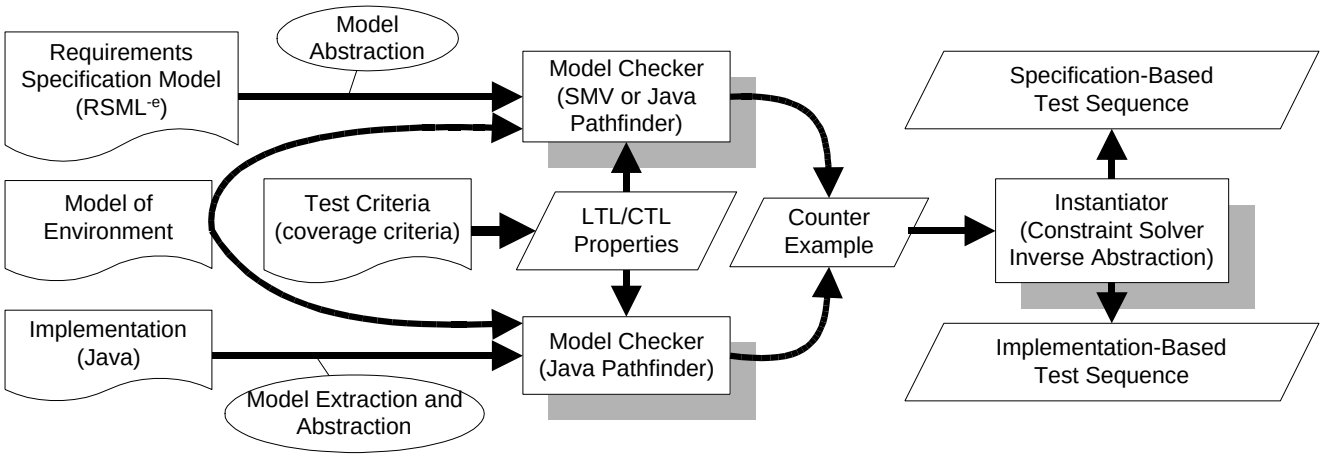

Figure 2. Test sequence generation overview and architecture.

criteria in terms of an underlying formal model, thus providing a uniform set of coverage criteria for a variety of specifications written in high-level specification languages. Initial results indicate that the approach will scale well to larger systems and have the potential to dramatically reduce the costs associated with generating test cases to high levels of structural coverage [15, 14].

The use of model-checkers as a test generation tool is being actively explored by some other researchers [2, 1, 8]. There is, however, much work to be done and several challenges need to be addressed in the future.

The problem of state space explosion can affect the search for counter-examples. Although the approach has worked remarkably well for us this far, state space explosion is a major concern when model-checking software artifacts and we expect to encounter this problem in the future. Often, specifications of software systems written in higher-level languages, such as SCR and RSML^{-e}, would include huge or infinite state-spaces that cannot be handled by model-checkers. The translation to a model checker must then abstract away enough detail to make model checking a feasible approach to test-case generation. Nevertheless, abstraction techniques that reduce the state-space could make instantiating the test case with concrete data somewhat difficult since the counter example would be presented in the abstract domain. We believe, however, that the very nature of test case generation with a model checker will lessen the state space explosion problem. After all, we are looking for a property that does not hold in the model and a counterexample is typically found very quickly. In the cases where a counterexample cannot be found (a transition that cannot be taken, MC/DC conditions that cannot be satisfied, etc.) we may have to explore the full state space. In the worst case, the verification run for the trap property can be terminated, and the property tagged for later manual analysis. In our limited pilot studies, we have not yet encountered this situation.

A goal in testing is to have a limited number of test cases without sacrificing test efficacy. In other words, one would prefer as small a test-suite as possible for a given test criteria. When generating test cases using our proposed method, where each test case is a sequence, a shorter test sequence could be subsumed by a longer one. In general two strategies can be adopted in reducing the size of the test suite generated by this approach. One way is to pre-process by combining trap properties for different test cases. In the case of, for example, MC/DC coverage, a condition may generate several redundant trap properties; one could compare trap properties and eliminate duplicates.

The other approach is to post-process the results. As soon as a test-case is generated one could execute the test sequence, measure the actual coverage on the artifact, and then remove all covered areas from further consideration for test-data generation. We could understand how this works in terms of program control flow. If a path to a decision point includes another decision point, a test case that exercises predicates in the former might also exercise certain predicates in the latter. Thus when generating test cases for the former, one could identify the tests that need not be generated for the latter. However, the task in specification-centered testing is to find an appropriate order of generating test cases so that longer test sequences are constructed before attempting to construct shorter sequences.

REFERENCES

[1] P. E. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of the Fourth*

IEEE International Symposium on High-Assurance Systems Engineering. IEEE Computer Society, Nov. 1999.

[2] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, Nov. 1998.

[3] M. R. Blackburn, R. D. Busser, and J. S. Fontaine. Automatic generation of test vectors for SCR-style specifications. In *Proceedings of the 12th Annual Conference on Computer Assurance, COMPASS'97*, June 1997.

[4] S. Burton, J. Clark, and J. McDermid. Testing, proof and automation. An integrated approach. In *Proceedings of First International Workshop on Automated Program Analysis, Testing and Verification*, June 2000.

[5] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transaction on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[6] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proc. of the 22nd Int'l Conf. on Software Engineering*, pages 439–448, June 2000.

[7] A. S. E. M. Clarke, E.A. Emerson. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, pages 244–263, April 1986.

[8] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.

[9] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[10] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

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

[12] S. Owre, N. Shankar, and J. Rushby. *The PVS Specification Language*. Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta relese edition, April 1993.

[13] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science*, pages 46–57, 1977.

[14] S. Rayadurgam and M. P. Heimdahl. Automated test-sequence generation from formal requirement models. Submitted to the 5th IEEE International Symposium on Requirements Engineering, February 2001.

[15] S. Rayadurgam and M. P. Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, pages 83–91. IEEE Computer Society, April 2001.

[16] J. Rushby. Model checking and other ways of automating formal methods. position paper for panel on Model Checking for Concurrent Programs; Software Quality Week, May/June 1995.

[17] J. M. Thompson, M. P. Heimdahl, and D. M. Erickson. Structuring formal control systems specifications for reuse: Surviving hardware changes. In *Proceedings of the Fifth NASA Langley Formal Methods Conference (Lfm2000)*, 2000.

[18] J. M. Thompson, M. P. Heimdahl, and S. P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.

[19] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedigns of the 14th IEEE International Automated Software Engineering Conference*, September 2000.

[20] M. W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.

[21] H. Zhu, P. Hall, and J. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.