Position paper for VSTTE, Zurich, Switzerland, October 2005.

Automated Test Generation And Verified Software *

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 USA

Abstract. Testing remains the principal means of verification in many certification regimes. Formal methods of verification will coexist with testing and should be developed in ways that improve, supplement, and exploit the value of testing. I describe automated test generation, which uses technology from formal methods to mechanize the construction of test cases, and discuss some of the research challenges in this area.

1 Introduction

By *testing* I mean observation of a program in execution under controlled conditions. Observations are compared against an explicit or informal oracle to detect bugs or confirm correctness. Much of the testing process is automated in modern development environments, but construction of test cases (i.e., the specific experiments to be performed) remains a largely manual process.

Testing is the method by which most software is verified today. This is true for safety critical software as well as the commodity variety: the highest level of flight critical software (DO-178B Level A) is required to be tested to a structural code coverage criterion known as MC/DC (Modified Condition/Decision Coverage) [1]. And although formal methods of analysis and verification are becoming sanctioned, even desired, by some certification regimes, testing continues to be required also—because it can expose different kinds of problems (e.g., compiler bugs), can examine the program in its system context, and increases the diversity of evidence available.

The weakness of testing is well-known to the formal methods and verification communities—it can only show the presence of bugs—but those communities are now beginning to recognize its strength: it *can* show the presence of bugs—often, very effectively. It is a great advantage in verification if the software to be verified is actually correct, so inexpensive methods for revealing incorrectness early in the development and verification process are necessary for verified software to be economically viable.

* This work was partially supported by NASA Langley Research Center through contract NAS1-00079, and by SRI International.

Thus, testing is not a rival to formal methods of verification, but a valuable and complementary adjunct. It is worthwhile to study how each can support the other, both in the technology that they employ, and in their contribution to the overall goal of cost-effective verification.

In this regard, the most significant recent development in testing has been the application of technologies from verification (notably, model-checking, SAT solving, and constraint satisfaction) to automate the generation of test cases. Automated test generation poses urgent opportunities and challenges: there are many technical challenges in achieving effective automation, there is a wealth of opportunity in the different ways that automated testing can be used, and there are serious implications for traditional certification regimes, and opportunities for innovative ones; there are also opportunities for theoretical research in the relationship between testing and verification, and for empirical inquiry into their pragmatic combination.

In this position paper, I briefly survey the topics mentioned above, and suggest research directions for the development and use of automated test generation in verification.

2 Technology for Automated Test Generation

Much of the process of test execution and monitoring is automated in modern software development practice. But the generation of test cases has remained a labor-intensive manual task. Methods are now becoming available that can automate this process.

A simple test-generation goal is to find an input that will drive execution of a (deterministic, loop-free) program along a particular path in its control flow graph. By performing symbolic execution along the desired path and conjoining the predicates that guard its branch points, we can calculate the condition that the desired test input must satisfy. Then, by constraint satisfaction, we can find a specific input that provides the desired test case. This method generalizes to find tests for other structural coverage criteria, and for programs with loops, and for those that are reactive systems (i.e., that take an input at each step). A major impetus for practical application of this approach was the realization that (for finite state systems) it can be performed by an off-the-shelf model checker: we simply check the property "always not P," where P is a formula that specifies the desired structural criterion, and the counterexample produced by the model checker is then the test case desired [2]. Different kinds of structural or specification-based tests can be generated by choosing suitable P.

Using a model checker to generate tests in this way can be very straightforward in model-based development, where we have an executable specification for the program that is in, or is easily translated to, the language of a model checker: the tests are generated from the executable specification, which then provides the oracle when these are applied to the generated program. There are many pragmatic issues in the selection of explicit-state, symbolic, or bounded model checkers for this task [3] and it is, of course, possible to construct special-

ized test generators that use the technology of model checking but customize it appropriately for this application.

The test generation task becomes more challenging when tests are to be generated directly from a low-level program description, such as C code, when the the path required is very long (e.g., when it is necessary to exhaust a loop counter), when the program is not finite state, and when nondeterminism is present.

When tests are to be generated directly from C code, or similar, it is natural to adopt techniques from software model checking. These seldom translate the program directly into the language of the model checker but usually first abstract it in some way. Predicate abstraction [4] is the most common approach, and discovery of suitable predicates is automated very effectively in the lazy-abstraction approach [5]. Abstractions for test generation are not necessarily the same as those used for verification. For the latter, the abstraction needs to be conservative (i.e., it should have more behaviors than the concrete program), whereas in the former case we generally desire that any test generated from the abstraction should be feasible in the concrete program (i.e., the abstraction may have fewer behaviors than the concrete program) [6]. This impacts the method for constructing the abstraction, and the choice of theorem proving or constraint satisfaction methods employed [7].

When very long test sequences are needed to reach a desired test target, it is sometimes possible to generate them using specialized model checking methods (e.g., those based on an ATPG engine [8]), or by generating the test incrementally (so that each subproblem is within reach of the model checker). Some of the most effective current approaches for generating long test sequences use combinations of methods. For example, random test generation rapidly produces many long paths through the program; to reach an uncovered test target, we find a location "nearby" (e.g., measured by Hamming distance on the state variables) that has been reached by random testing and then use model checking or constraint satisfaction to extend the path from that nearby location to the one desired [9]. An alternative approach is to reduce the size of the model that represents the program (it is easier to find longer paths in smaller models): this can be done by standard model checking reductions such as slicing and cone of influence reduction, and also by the predicate abstraction techniques mentioned above.

Traditional model checking technology must be extended or adapted when the program is not finite state. In some cases, an infinite state bounded model checker can be used (i.e., a bounded model checker that uses a decision procedure for satisfiability modulo theories (SMT) [10] rather than a Boolean SAT solver) [11]. In other cases, such as those where inputs to the program are complex data structures (e.g., trees represented as linked lists), we can randomly or exhaustively generate all inputs up to some specified size. Straightforward approaches can be very inefficient (e.g., very few randomly generated list structures represent a valid red-black tree) and redundant (i.e., they generate many inputs that are structurally "isomorphic" to each other), so that it is best to

view the search as a constraint satisfaction problem and to use technology from that domain [12].

The test generation problem changes significantly when the program under test is nondeterministic, or when part of the testing environment is not under the control of the tester (e.g., testing an embedded system in its operational environment). In these cases, we cannot generate test sequences independently of their actual execution: it is necessary to observe the behavior of the system in response to the test generated so far and to generate the next input in a way that advances the purpose of the test. Thus, test generation becomes a problem of controller synthesis; methods for solving this problem can use technology similar to model checking but can seldom use an off-the-shelf model checker [13].

The problem becomes yet more difficult when the test environment includes mechanical systems: for example, testing the shift controller of an automatic gearbox in its full system context with a (real or simulated) gearbox attached. Here, the test generation problem is escalated to one of controller synthesis in a hybrid system (i.e., one whose description includes differential equations). This is a challenging problem, but a plausible approach is to replace the hybrid system elements of the modeled environment by conservative discrete approximations, and then use methods for test generation in nondeterministic systems [14]. As in the case of predicate abstraction, the notion of "conservative" that is suitable for test generation may differ from that used in verification.

3 Selection of Test Targets

The previous section has sketched how test cases can be generated automatically; the next problem is to determine how to make good use of this capability. One approach uses test generation to help developers explore their emerging designs [15]: a designer might say "show me a run that puts control at this point with $x \leq 0$." This approach is very well-suited to model-based design environments (i.e., those where the design is executable), but is less so for traditional programming. An approach that has proven useful in traditional programming is random test generation at the unit level. In some programming environments, each unit is automatically subjected to random testing against desired properties if these have been specified, or generic ones (e.g., no exceptions) as it is checked in (Haskell QuickCheck [16] is the progenitor of this approach). A similar approach can be used in theorem proving environments: before attempting to prove a putative theorem, first try to refute it by random test generation [17] (in PVS, this can also be tried during an interactive proof, if the current proof goal looks intractable). These simple approaches are highly effective in practice. More challenging tests can be achieved by exhaustive generation of inputs up to some bounded size [18]. In Extreme Programming, tests take on much of the rôle played by specifications in more traditional development methods [19], and automated, incremental test generation can support this approach [20].

More traditional uses of testing are for systematic debugging, and for validation and verification. In tests developed by humans, the first of these is generally

driven by some explicit or implicit hypotheses about likely kinds of bugs, while the others are driven by systematic "coverage" of requirements and code.

One simple fault hypothesis is that errors are often made at the boundaries of conditions (e.g., the substitution of $<$ for $\leq$) and some automated test generators target these cases [21]. Another hypothesis is that compound decisions (e.g., $A \wedge B \vee C$) may be constructed incorrectly so tests should target the "meaningful impact" [22] of each condition within the decision (i.e., each must be shown able to independently affect the outcome).[1] It turns out that these ideas are related: boundary testing for $x \leq y$ is equivalent to rewriting the decision as $x < y \vee x = y$ and then testing for meaningful impact of the two conditions. The classes of faults detected by popular test criteria for compound decisions have been analyzed by Kuhn [23] and extended by others [24, 25].

Requirements- or specification-based testing is most easily automated when the requirements or specification are provided in executable form—as is commonly done in model based development. Here, we can use the methods sketched in Section 2 to generate tests that explore portions of the specified behavior. The usual idea is that a good set of tests should thoroughly explore the control structure of the specification; typical criteria for such structural coverage are to reach every control state, to take every transition between control states, and more elaborate variants that explore the conditions within the decisions that control selection of transitions (as in the meaningful impact criteria mentioned earlier). Structural coverage criteria can be augmented by "test purposes" [26] that describe the kind of tests we want to generate (e.g., those in which the `gear` input to a gearbox shift selector changes at each step, but only to an adjacent value), or by predicates that describe relationships that should be explored (e.g., a queue is empty, full, or in between) [27]. Test purposes and predicates are related to predicate abstraction and can be used to reduce the statespace of the model, and thereby ease the model checking task underlying the test generation. Generating a separate test for each coverage target produces inefficient test sets that contain many short tests and much redundancy, so recent methods attempt to construct more efficient "tours" that visit many targets in each test [3, 27].

Requirements-based testing is more difficult when requirements are specified as properties. One approach is to translate the properties into automata (i.e., synchronous observers), then target structural coverage in the automata.

4 Testing for Verification

Certification regimes for which testing is an important component generally require evidence that the testing has been thorough. DO-178B Level A (which applies to the highest level of flight-critical software in civil aircraft) is typical: it requires MC/DC code coverage. The expectation is that tests are generated by consideration of requirements and their execution is monitored to measure coverage of the code. As the industry moves toward model-based development,

[1] This use of *decision* and *condition* is the one employed in MC/DC, which is a testing criterion of this kind.

it can be argued that the requirements are represented by the models, and hence that automated test generation from the model is a form of requirements-based testing. One way to do this is by targeting MC/DC coverage in the model. Heimdahl, George, and Weber did this for a model of a flight guidance system developed by Rockwell, and then executed the tests on implementations that had been seeded with errors [28]. They found that the autogenerated tests detected relatively few bugs, and generally performed worse than random testing. Part of the explanation for this distressing observation is that the model checking technology underpinning the test generation is "too clever": it generally finds the *shortest* test to discharge any given goal, and these short tests often exploit some special case and never reach the interesting parts of the state space. There is hope that methods that generate tours through many test goals will do better than those that target the goals individually, or that suitable test purposes may guide the test generator into more productive areas of the state space, but these ideas need to be validated in practice.

Another way in which testing has been employed for verification is in "conformance testing," which is generally applied to distributed systems and protocols. Given a formal specification and an implementation that purports to satisfy it, conformance testing generates a series of tests such that any departure from the specification will eventually be revealed (subject to various technical caveats) [29]. Only a relatively small number of tests can be performed in practice, so the eventuality guarantee is of mainly theoretical interest, and the more pragmatic concern is to try and arrange things so that tests generated early in the series are effective at finding bugs.

There is relatively little work that combines automated testing with formal verification. One attractive approach developed by Rusu uses test generation to decompose the classical formal verification problem into smaller components [30].

5 Research Challenges

Testing is the dominant means of verification used today. Any research agenda in software verification must include testing as a topic, and its roadmap must suggest how the proposed research will improve testing, and how it can use it, as well as how it may replace it in selected areas.

Automated test generation is an attractive topic in this area: it can reduce the cost of testing and may improve its quality. And it is an "invisible" application of formal methods and thus provides a good opportunity to introduce this technology to new communities. Among the most eager adopters of this capability are those in regulated industries where onerous testing requirements constitute a significant part of overall development costs. As mentioned above, there is some evidence that simply using the test coverage requirements as a target for automated test generation may be a flawed strategy: coverage metrics are intended to measure the thoroughness of human-generated tests, and do not necessarily lead to good test sets when used in an inverted role as a specification for the tests required.

Thus, an urgent research topic is development of techniques for specifying good test sets. There are two subtopics here: the role of the human tester will change from construction of tests to *specification* of tests (the tests will be generated automatically from the specification), so we need ideas and techniques for specifying tests (e.g., an extended notion of test purpose); second, we need empirical data on what kinds of test specification produce good tests (i.e., those that are effective in revealing errors). Because automated test generation performs constraint satisfaction (either explicitly, or implicitly via model checking), it is possible to specify test purposes using a recognizer rather than a generator, and this creates attractive possibilities [31].

Most current methods and tools for automated test generation are limited to unit tests. A second general research area is development of methods and technology for other (arguably more important) testing tasks, such as integration and system tests. At these levels, tests become interactive programs, and the formal context becomes that of controller synthesis for nondeterministic, timed, and hybrid systems. Abstraction is likely to be necessary, both for the system under test and for its environment, and there are interesting questions regarding the appropriate kinds of abstractions to use, and the theorem proving and model checking methods that are most suitable for constructing and using them.

A third suggested general research area is the integration of testing with formal methods of analysis and verification. Again, there are two subtopics: one is technical integration—for example, how can testing help in formal specification and proof (cf. QuickCheck-like methods for rapid refutation)—while the other focuses on how the overall verification process can be decomposed into elements that are effectively tackled by different means. There are proposals, for example, to replace some unit test requirements in avionics by static analysis; yet testing can address some issues (such as compiler bugs, which are a genuine problem) that static analysis does not (unless applied to machine code), so the overall web of argument in support of verification may become interestingly complex. A companion paper in these proceedings outlines some of the issues in technical integration of verification components [32], while the larger issues of "compositional assurance," in which the assurance case for a system is composed from different kinds of verification evidence for its components, is only just beginning to receive attention.

References

1. Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J., Rierson, L.K.: A practical tutorial on modified condition/decision coverage. NASA Technical Memorandum TM-2001-210876, NASA Langley Research Center, Hampton, VA (2001) Available at `http://www.faa.gov/certification/aircraft/av-info/software/Research/MCDC%20Tutorial.pdf`.
2. Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. In Nierstrasz, O., Lemoine, M., eds.: Software Engineering—ESEC/FSE '99: Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering. Volume

1687 of Lecture Notes in Computer Science., Toulouse, France, Springer-Verlag (1999) 146–162
3. Hamon, G., de Moura, L., Rushby, J.: Generating efficient test sets with a model checker. In: 2nd International Conference on Software Engineering and Formal Methods, Beijing, China, IEEE Computer Society (2004) 261–270
4. Saïdi, H., Graf, S.: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: Computer-Aided Verification, CAV '97. Volume 1254 of Lecture Notes in Computer Science., Haifa, Israel, Springer-Verlag (1997) 72–83
5. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with Blast. In: Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN). Volume 2648 of Lecture Notes in Computer Science., Springer-Verlag (2003) 235–239
6. Ball, T., Kupferman, O., Yorsh, G.: Abstraction for falsification. [33] 67–81
7. Xia, S., Di Vito, B., Muñoz, C.: Toward automated test generation for engineering applications via predicate abstraction. In: 20th IEEE International Conference on Automated Software Engineering (ASE'02), Long Beach, CA, IEEE Computer Society (2005) To appear.
8. Boppana, V., Rajan, S.P., Takayama, K., Fujita, M.: Model checking based on sequential ATPG. [34] 418–430
9. Ho, P.H., Shiple, T., Harer, K., Kukula, J., Damiano, R., Bertacco, V., Taylor, J., Long, J.: Smart simulation using collaborative formal simulation engines. In: International Conference on Computer Aided Design (ICCAD), Jan Jose, CA, Association for Computing Machinery (2000) 120–126
10. Barrett, C., de Moura, L., Stump, A.: SMT-COMP: Satisfiability modulo theories competition. [33] 20–23
11. de Moura, L., Rueß, H., Sorea, M.: Lazy theorem proving for bounded model checking over infinite domains. In Voronkov, A., ed.: 18th International Conference on Automated Deduction (CADE). Volume 2392 of Lecture Notes in Computer Science., Copenhagen, Denmark, Springer-Verlag (2002) 438–455
12. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: International Symposium on Software Testing and Analysis (ISSTA), Rome, Italy, Association for Computing Machinery (2002) 123–122
13. Jéron, T., Morel, P.: Test generation derived from model-checking. [34] 108–121
14. Tiwari, A.: Abstractions for Hybrid Systems, Computer Science Laboratory, SRI International, Menlo Park, CA. (2004) Combines several conference papers: available at `http://www.csl.sri.com/~tiwari/new.pdf`.
15. Ben-David, S., Gringauze, A., Sterin, B., Wolfsthal, Y.: PathFinder: A tool for design exploration. In: Computer-Aided Verification, CAV '2002. Volume 2404 of Lecture Notes in Computer Science., Copenhagen, Denmark, Springer-Verlag (2002) 510–514
16. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: International Conference on Functional Progamming, Montreal, Canada, Association for Computing Machinery (2000) 268–279
17. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: 2nd International Conference on Software Engineering and Formal Methods, Beijing, China, IEEE Computer Society (2004) 230–239
18. Sullivan, K., Yang, J., Coppit, D., Khurshid, S., Jackson, D.: Software assurance by bounded exhaustive testing. In: International Symposium on Software Testing and Analysis (ISSTA), Boston, MA, Association for Computing Machinery (2004) 133–142

19. Beck, K.: Test Driven Development: By Example. Addison-Wesley (2002)
20. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.: Extreme model checking. In: Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday. Volume 2772 of Lecture Notes in Computer Science., Springer-Verlag (2004) 332–358
21. Kosmatov, N., Legeard, B., Peureux, F., Utting, M.: Boundary coverage criteria for test generation from formal models. In: 15th International Symposium on Software Reliability Engineering (ISSRE'04), Saint-Malo, France, IEEE Computer Society (2004) 139–150
22. Weyuker, E., Goradia, T., Singh, A.: Automatically generating test data from a Boolean specification. IEEE Transactions on Software Engineering **20** (1994) 353–363
23. Kuhn, D.R.: Fault classes and error detection capability of specification-based testing. ACM Transactions on Software Engineering and Methodology **8** (1999) 411–424
24. Tsuchiya, T., Kikuno, T.: On fault classes and error detection capability of specification-based testing. ACM Transactions on Software Engineering and Methodology **11** (2002) 58–62
25. Okun, V., Black, P.E., Yesha, Y.: Comparison of fault classes in specification-based testing. Information and Software Technology **46** (2004) 525–533
26. Clarke, D., Jéron, T., Rusu, V., Zinovieva, E.: STG: a symbolic test generation tool. In Katoen, J.P., Stevens, P., eds.: Tools and Algorithms for the Construction and Analysis of Systems: 8th International Conference, TACAS 2002. Volume 2280 of Lecture Notes in Computer Science., Grenoble, France, Springer-Verlag (2002) 470–475
27. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: International Symposium on Software Testing and Analysis (ISSTA), Rome, Italy, Association for Computing Machinery (2002) 112–122
28. Heimdahl, M.P., George, D., Weber, R.: Specification test coverage adequacy criteria = specification test generation *In*adequacy criteria? In: High-Assurance Systems Engineering Symposium, Tampa, FL, IEEE Computer Society (2004) 178–186
29. Tretmans, J., Belinfante, A.: Automatic testing with formal methods. In: EuroSTAR'99: 7^{th} European Int. Conference on Software Testing, Analysis & Review, Barcelona, Spain, EuroStar Conferences, Galway, Ireland (1999)
30. Rusu, V.: Verification using test generation techniques. In Eriksson, L.H., Lindsay, P., eds.: Formal Methods Europe (FME'02). Volume 2391 of Lecture Notes in Computer Science., Copenhagen, Denmark, Springer-Verlag (2002) 252–271
31. Hamon, G., de Moura, L., Rushby, J.: Automated test generation with SAL. Technical note, Computer Science Laboratory, SRI International, Menlo Park, CA (2004) Available at `http://www.csl.sri.com/users/rushby/abstracts/sal-atg`.
32. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N.: Integrating verification components. These proceedings (2005)
33. Etessami, K., Rajamani, S.K., eds.: Computer-Aided Verification, CAV '2005. Volume 3576 of Lecture Notes in Computer Science., Edinburgh, Scotland, Springer-Verlag (2005)
34. Halbwachs, N., Peled, D., eds.: Computer-Aided Verification, CAV '99. Volume 1633 of Lecture Notes in Computer Science., Trento, Italy, Springer-Verlag (1999)