TOWARDS TEST CASES GENERATION FROM SOFTWARE SPECIFICATIONS

Mrs.R. Jeevarathinam
Assistant Professor,
Department of Computer Science
SNR Sons College,
Coimbatore-06.

Dr. Antony Selvadoss Thanamani
Professor and Head,
Department of Computer Science
NGM College, Pollachi.

Abstract

Verification and Validation of software systems often consumes up to 70% of the development resources. Testing is one of the most frequently used Verification and Validation techniques for verifying systems. Many agencies that certify software systems for use require that the software be tested to certain specified levels of coverage. Currently, developing test cases to meet these requirements takes a major portion of the resources. Automating this task would result in significant time and cost savings. This testing research is aimed at the generation of such test cases. In the proposed approach a formal model of the required software behavior (a formal specification) is used for test-case generation and as an oracle to determine if the implementation produced the correct output during testing. This is referred to as Specification Based Testing. Specification based testing offers several advantages to traditional code based testing. The formal specification can be used as the source artifact to generate functional tests for the final product and since the test cases are produced at an earlier stage in the software development, they are available before the implementation is completed. Central to this approach is the use of model checkers as test case generation engines. Model checking is a technique for exploring the reachable state-space of a system model to verify properties of interest. There are several research challenges that must be addressed to realize this test generation approach.

Keywords: Test case generation; software; specifications; testing

1. Introduction

The purpose of software engineering is to build high quality software. The systematic production of high quality software, which meets its specification, is still a major problem. Although formal specification methods have been around for a long time, only a few safety-critical domains justify the erroneous effort of their application. The testing has been more highlighted in software life cycle. Although good program design and programming practice may increase the correctness of software, the testing is the only way to guarantee that the software meets its requirements properly.

A program is tested by manually creating a test suite. In Figure 1 The Trace Abstractor takes as input a model of the input application program to be tested. The model furthermore describes a mapping from input values to usage rules for each input element the model defines what rules an execution on that input should satisfy. The Trace Abstractor generates traces to the application program. For each generated trace input, a set of test cases is generated. The observer module checks the trace against the guaranteed set of rules. Therefore it takes the execution trace as input and produces the set of generated rules as output. This process in some cases can be automated. A special characteristic is that the rules to be verified are generated automatically from the inputs to the program to be tested.

The rest of the paper is organized as follows. Section 2 outlines the motivation of this work. Section 3 describes about the related work. Section 4 gives the various technologies for test case generation. Section 5 discusses about the automatic test case generation. Section 6 concludes and outlines how this work will be continued in the future.

2. Motivation

There are two reasons for an input that causes the test cases to violate its specifications. They are: First, Improper or even no specification is available, either because no one is capable of writing it or because it is not expressible in the specified language. Writing a correct logical specification is a difficult task that often requires specialized skilled thinking. In practice, the programmer may find easier to manually inspect a series of test executions for correctness than to write a correct, general specification for all possible executions. Second, even in the presence of a specification, the cost of statically checking the code against the specification may be far greater than running the test suite. If so, it may be profitable to invest substantial time up front to generate a regression test suite that can be run in the future.

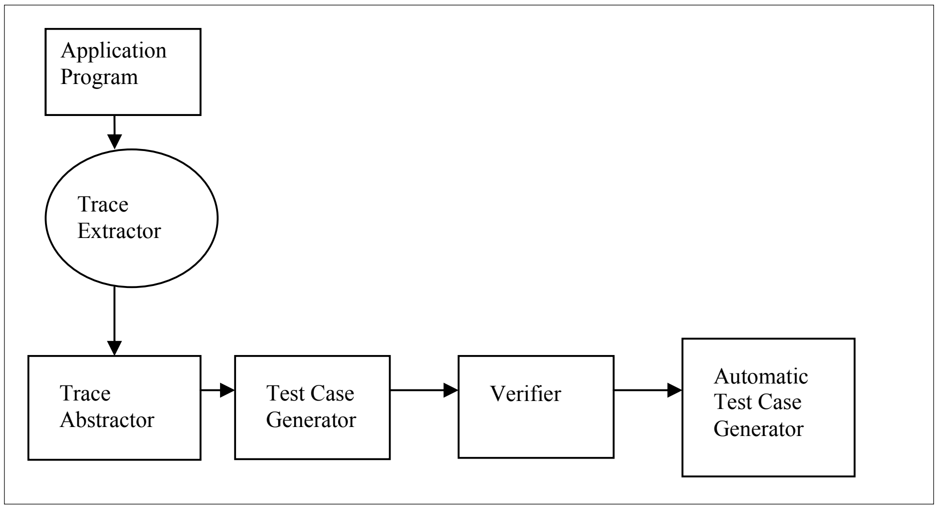


Fig. 1: Test Case Generation and Verification

3. Related Work

Constraints are used to represent inputs [16,7,18,26] and has been implemented in various tools including EFFIGY [18], TEGTGEN [20] and INKA [35]. Most of this work has been focused on solving constraints on primitive data, like Booleans and integers.

Some latest frameworks, such as TestEra [22] and Korat [34,21], do support generation of complex structures. In TestEra inputs are generated from constraints given in Alloy, which is a first-order declarative language based on relations and sets. TestEra also uses off-the-shelf SAT solvers for solving constraints. In Korat inputs are generated from constraints given as Java predicates. The Korat algorithm has recently been included in the AsmL Test Generator [9] to enable generation of structures. TestEra and Korat focus on solving structural constraints. They do not directly solve constraints on primitive data. Instead, they systematically try all primitive values within given bounds, which may be inefficient.

AsmLT's first version named Test Generator [11] was based on finite-state machines (FSMs): an AsmL [13] specification is transformed into an FSM and different traversals of the FSM are used to construct test inputs. In Korat structure generation is added based on finite-state machines [11]. AsmLT was successfully used in the faults detection in a production-quality XPath compiler [28].

The use of model checking for test input generations have investigated by several researchers (see [15] for a good survey). Gargantini and Heitmeyer [10] generate tests that use a model checker to violate known properties of a specification given in the SCR notation. Ammann and Black [4] combine mutation analysis and model checking to generate test cases from a specification. Rayadurgam et al. use a structural coverage-based approach to generate test cases from specifications given in RSML−e by using a model checker [14]. Hong et al. formulate a theoretical framework for using temporal logic to specify data-flow test coverage in [15]. These approaches cannot handle structurally complex inputs.

Nowadays there are several tools available which produce test inputs from a description of tests. For testing Haskell programs the tool namely QuickCheck [33] is used. It requires the tester to write Haskell functions which can produce valid test inputs; executions of such functions with different random seeds produce different test inputs. The proposed work differs in that it requires only a specification that characterizes valid test inputs and then uses a general purpose search to generate all valid inputs up to a certain size. DGL [23] and lava [27] generate test inputs from production grammars that were used for random testing, although they can also generate test inputs. However, they cannot easily represent inputs with complex structure, which can be examined by using Java as a specification language.

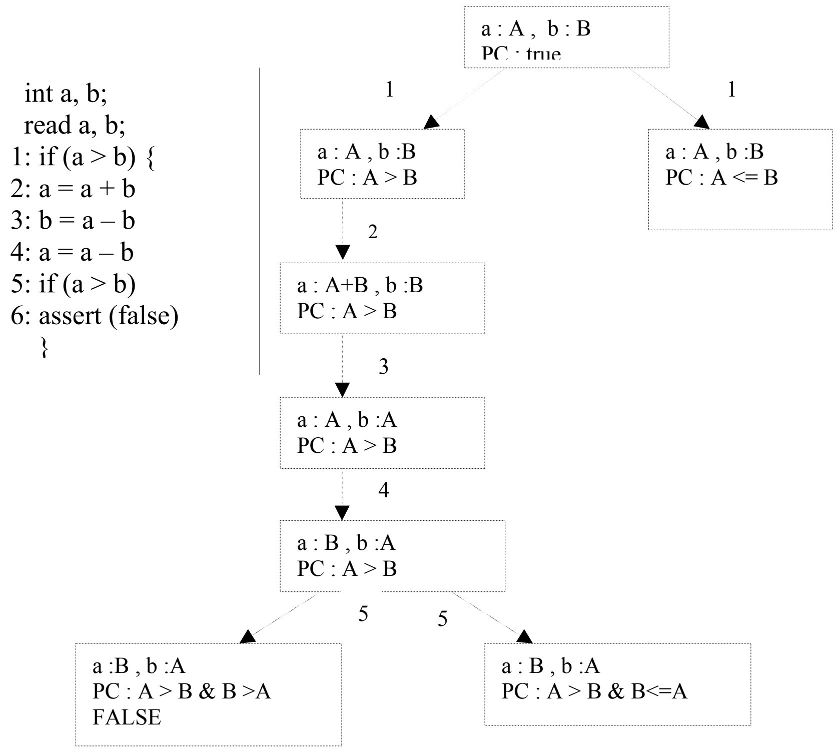
4. Test Case Generation

4.1 ***Model-based testing***

Latest researchers found that, the generation of test inputs and test cases for a program which is under test is a time consuming and tedious manual activity. But, test input generation and test cases generation leads itself to automation and therefore has been the focus of much researchers attention – recently it has also been adopted in industry [24,32,8,11]. There are two main approaches for generating automatic test inputs namely: a static approach and a dynamic approach. Static approach generates inputs from some available kind of model of the system which is also called as model-based testing. But dynamic approach generates tests by executing the program repeatedly, while employing criteria to rank the quality of the tests produced [19,29]. The dynamic approach is based on the observation that test input generation can be seen as an optimization problem, where the cost function used for optimization is typically related to code coverage, e.g. statement or branch coverage. The model-based test data input and the e generation of test cases approach is more commonly used. AGEDIS tool [1] is used for automated generation and execution of test suites for distributed component based software, TGV tool [2] is used for the generation of conformance test suites for protocols; refer also Hartman’s survey of the field [31]. For model-based testing the sample model can behave like a model of expected system behavior and it can be derived from a number of sources, like, a model of the requirements, use cases, design specifications of a system [31] (i.e) even the code itself can also be used to create a model based on its symbolic execution for various input conditions [18,24]. In dynamic approach, it is most typical to use some notion of coverage of the model to derive test inputs, i.e., generate inputs that cover all transitions, or branches, etc., in the model. Constructing a model of the expected system behavior can be a costlier process. On the other hand, generating test inputs just based on a specification of the input structure and input pre-conditions can be very effective, while typically less costly. This is the approach explained in the following.

In [17] a framework is presented that combines symbolic execution and model checking techniques for the verification of Java programs. The framework can be used for test input generation for white-box and black-box testing. For white-box test input generation, the framework model checks the program under test. A testing coverage criterion, e.g. branch coverage, is encoded in a temporal logic specification. Counterexamples to the specification represent paths that satisfy the coverage criterion. During model checking symbolic execution is performed which computes a set of constraints of all the inputs that execute those paths. Appropriate testing is needed to solve the input data constraints in order to instantiate test inputs which can be executed. The above mentioned framework can also be used for black-box test input data generation. In this type of testing, the inputs to the program are described as a Java input specification, i.e., a Java program, annotated with special instructions to encode constraints, for symbolic execution. The framework is then used to check this Java specification, i.e., to systematically explore the input domain of the program under test and to generate inputs according to this specification. But for black-box test input data generation, only the input specification is required to be expressed in Java; the program can be written in any another language such as C++ . While writing input specifications, it is possible to express inputs with complex structure like linked lists, red-black search trees, and executive plans.

For test input data generation symbolic execution is a popular approach, but it can handle only sequential code with simple data. This technique has been extended in [17], to handle complex data structures such as linked lists and trees, concurrency as well as linear constraints on integer data. Symbolic execution of a program path results in a set of constraints that can define program inputs that execute the path. These constraints are then solved using decision procedures to generate concrete test inputs. When the program represents an executable input specification, symbolic execution of the specification enables to generate inputs that give, full specification coverage. Note that these specifications are typically not very large – no more than a few thousand lines, and it will allow efficient symbolic execution.



```
 int a, b;
 read a, b;
1: if (a > b) {
2: a = a + b
3: b = a – b
4: a = a – b
5: if (a > b)
6: assert (false)
   }
```

Fig.2. Code for swapping integers and corresponding symbolic execution tree.

4.2 ***Symbolic Execution of specifications***

For black-box test input generation from a specification the symbolic execution concept is used as it can be applied for white box testing. The main idea behind this kind of symbolic execution [18] is to use symbolic values, instead of actual data, as input values and to represent the values of program variables as symbolic expressions. The state of a symbolically executed program includes, program variables and the program counter. Here in addition to the symbolic values of variables, a path condition is also included. The path condition is a Boolean formula used for the symbolic inputs; it accumulates constraints. These constraints of the inputs must satisfy because the program execution must follow the particular associated path. A symbolic execution tree characterizes the execution paths followed during the symbolic execution of a program in which the nodes represent program states and the arcs represent transitions between states.

In [17] the example swaps the values of integer variables a and b when a is greater than b. Figure 2 is the corresponding tree representation of the symbolic execution. Initial value of the path condition, PC, is true and the variables a and b have symbolic values A and B, respectively. At each branching condition, PC is updated with assumption values about the input data according to the alternative possible paths. For example, when the first statement is executed, both then and else alternatives of the "if" statement were possible and PC is updated accordingly. If the path condition PC is false, (i.e., there is no set of inputs that satisfy it), this means that the symbolic state of the input specification is not reachable and symbolic execution does not continue in that path. In the above mentioned example, statement (6) is unreachable. To find a test input data to reach branch statement (5) it is necessary to solve constraint A > B, for example make inputs a and b as 1 and 0, respectively. starts the symbolic execution of a procedure on uninitialized inputs and it assigns values to these inputs, i.e., it initializes parameters when they are first accessed during symbolic execution of the procedure. This allows symbolic execution of procedures without requiring a priori bound on the number of input objects. Procedure preconditions are used to initialize inputs only with valid values.

In the context of sequential programs with a fixed number of integer variables the concept of symbolic execution arose. This technique [17] have been extended to handle dynamically allocated data structures like linked lists, trees, complex preconditions like lists that have to be acyclic, other primitive data such as strings, and concurrency. A key feature of the proposed algorithm is that it starts the symbolic execution of a procedure on uninitialized inputs and it assigns values to these inputs, i.e., it initializes parameters when they are first accessed during symbolic execution of the procedure. This allows symbolic execution of procedures without requiring a priori bound on the number of input objects. Procedure preconditions are used to initialize inputs only with valid values.

5. Automatic Test Case Generation

The symbolic execution-based framework can handle all of the language features of Java and in addition treats non-deterministic choice expressed in annotations of the program being analyzed. This has been extended with a symbolic execution capability which is described in detail in [17]. Figure 3 illustrates the framework for test input data generation. The input specification is given as a non-deterministic program that is instrumented to add support for manipulating formulas that represent each path conditions. The instrumentation enables to perform symbolic execution. Essentially, the model checker explores the symbolic state space of the program, for example, the symbolic execution tree in Figure 2. A symbolic state includes information about the heap configuration and the path condition on integer variables. Whenever a path condition is changed, it is checked for correctness using an appropriate decision procedure. If the path condition is found to be erroneous, the model checker backtracks and takes other available paths of execution. A testing coverage criterion is added and encoded in the property which the model checker should check for. This directs the model checker to create an alternate example trace whenever a valid symbolic test input has been generated.

From this trace of execution a test input is generated. Among the variables and constraints only input variables are allowed to be symbolic, all constraints are described in terms of inputs. Finding a solution to these constraints will allow a valid set of test data to be produced. In future work the solution discovery process will be refined to consider characteristics such as boundary condition cases.

The model checker is not required to perform state matching because it is not decided when states represent path conditions on unbounded data. But performing symbolic execution on programs with loops can explore infinite number of execution trees, so symbolic execution might not terminate its process. Therefore, for systematic state space exploration, only limited depth-first search or breadth-first search is used; this framework also supports different search strategies such as branch coverage [12] or random search.

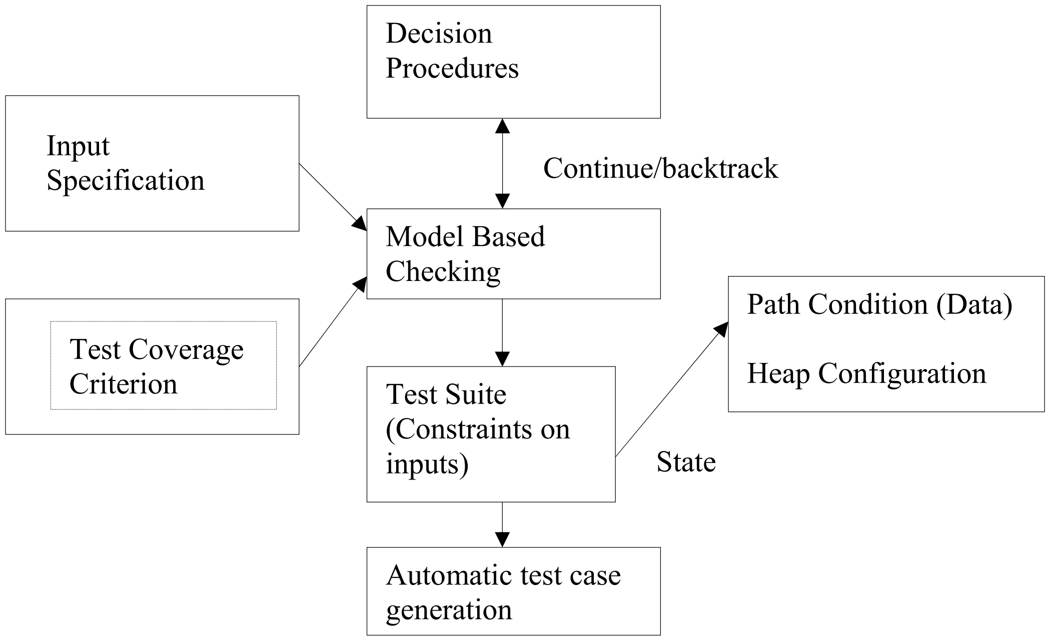


Fig.3: Automatic Test Case Generation

6. Conclusions and Future Work

A framework for testing based on automated test case generation using input specification has been presented. This paper proposed and demonstrated the use of model checking and symbolic execution for test case generation and the use of temporal logic monitoring during the execution of the test cases. The framework requires construction of a trace extractor, trace abstractor and a test case generator for the application. From that, a large test suite can be automatically generated, executed and verified to be in conformity with the properties. For each input (generated by the test input generator) the property generator constructs a set of properties that must hold when the program under test is executed on that input. The program is instrumented to emit an execution log of events. The Verifier checks that the event log satisfies the set of properties. Writing test oracles as temporal logic formulas is both natural and leverages algorithms that efficiently check if execution on a test input conforms to the properties. While property definition is often difficult, at least for some domains, an effective approach is to write a property generator, rather than a universal set of properties that are independent of the test input. Note also that the properties need not completely characterize correct execution. Instead, a user can choose among a spectrum of weak but easily generated properties to strong properties that may require construction of complex formulas. In the near future, exploring techniques to improve the quality of the generated test suite by altering the search strategy of the model checker by improving the symbolic execution technology will be concentrated. It will also investigate improvements to the logic and its engine. In particular an attempt will be made to integrate the concurrency analysis algorithms. This work is continuing instrumentation of Java byte code and will extend this work to C and C++. Some other research group has done fundamental research in other areas, such as software model checking (model checking the application itself and not just the input domain) and static analysis. In general, the ultimate goal is to combine the different technologies into a single coherent framework.

References

[1] AGEDIS - model based test generation tools. http://www.agedis.de.
[2] The test sequence generator TGV. http://www-verimag.imag.fr/~async/TGV.
[3] *1st, 2nd, 3rd and 4th Workshops on Runtime Verification (RV'01 - RV'04)*, volume 55(2), 70(4), 89(2), (RV'04 to be published) of *ENTCS*. Elsevier Science: 2001, 2002, 2003, 2004. http://ase.arc.nasa.gov/rv2004.
[4] P. Ammann and P. Black. A Specification-Based Coverage Metric to Evaluate Test Sets. In *Proceedings of the 4th IEEE International Symposium on High Assurance Systems and Engineering*, 1999.
[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, July 2002.
[6] K. Claessen and J. Hughes. Testing Monadic Code with QuickCheck. In *Proc. ACM SIGPLAN workshop on Haskell*, pages 65–77, 2002.
[7] L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, September 1976.
[8] D. Drusinsky. The Temporal Rover and the ATG Rover. In *Proc. of SPIN'00: SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330, Stanford, California, USA, 2000. Springer.
[9] Foundations of Software Engineering, Microsoft Research. The AsmL test generator tool. http://research.microsoft.com/fse/asml/doc/AsmLTester.html.
[10] A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In *Proceedings of the 7th European engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 146–162. Springer-Verlag, 1999.
[11] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating Finite State Machines from Abstract State Machines. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 112–122, July 2002.
[12] A. Groce and W. Visser. Model Checking Java Programs using Structural Heuristics. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA)*, pages 12 – 21. ACM Press, July 2002.
[13] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
[14] M. P. E. Heimdahl, S. Rayadurgam, W. Visser, D. George, and J. Gao. Auto- Generating Test Sequences using Model Checkers: A Case Study. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software (FATES)*, Montreal, Canada, October 2003.

[15] H. Seok Hong, I. Lee, O. Sokolsky, and H. Ural. A Temporal Logic Based Theory of Test Coverage and Generation. In *Proc. 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Grenoble, France, April 2002.
[16] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3), 1975.
[17] S. Khurshid, C. Pasareanu, andW. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of TACAS'03: Tools and Algorithms for theConstruction and Analysis of Systems*, volume 2619 of *LNCS*, Warsaw, Poland, April 2003.
[18] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
[19] B. Korel. Automated Software Test Data Generation. *IEEE Transaction on Software Engineering*, 16(8):870–879, August 1990.
[20] B. Korel. Automated Test Data Generation for Programs with Procedures. San Diego, CA, 1996.
[21] D. Marinov. *Testing Using a Solver for Imperative Constraints*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004. (to appear).
[22] D. Marinov and S. Khurshid. TestEra: A Novel Framework for Automated Testing of Java Programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, November 2001.
[23] P. M. Maurer. Generating Test Data with Enhanced Context-Free Grammars. *IEEE Software*, 7(4):50–55, July 1990.
[24] Parasoft. http://www.parasoft.com.
[25] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102 – 114, August 1992.
[26] C. V. Ramamoorthy, Siu-Bun F. Ho, and W. T. Chen. On the Automated Generation of Program Test Data. *IEEE Transactions on Software Engineering*, 2(4), 1976.
[27] E. G. Sirer and B. N. Bershad. Using Production Grammars in Software Testing. In *Proc. 2nd conference on Domain-specific languages*, pages 1–13, 1999.
[28] K. Stobie. Advanced Modeling, Model Based Test Generation, and Abstract state machine Language AsmL. http://www.sasqag.org/pastmeetings/asml.ppt, 2003.
[29] N. Tracey, J. Clark, and K. Mander. The Way Forward for Unifying Dynamic Test-Case Generation: The Optimisation-Based Approach. In *International Workshop on Dependable Computing and Its Applications (DCIA)*, pages 169–180. IFIP, January 1998.
[30] W. Visser, K. Havelund, G. Brat, S.-J. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2), April 2003.
[31] A. Hartman. Model Based Test Generation Tools. http:// www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf.
[32] T-VEC. http://www.t-vec.com.
[33] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *Proc. of CAV'04: Computer Aided Verification,*Lecture Notes in Computer Science. Springer-Verlag, 2004.
[34] C. Artho, K. Havelund, and A. Biere. High-level Data Races. In *VVEIS'03: The First International Workshop on Verification and Validation of Enterprise Information Systems*, France, April 2003.
[35] A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation using Constraint Solving Techniques. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Clearwater Beach, FL, 1998.

Mrs R. Jeevarathinam graduated with MCA in 2001 from Bharathiar University, India and completed M.Phil from Bharathidasan University, India during 2003-04. Her areas of Interest include Software Engineering & Data Mining. She has about 8 years of teaching experience. Currently she is working as a Sr. Lecturer in CSE at SNR Sons College, Coimbatore, India and also pursuing PhD of Mother Teresa Women University, India. She has published a number of papers in various national & international journals & conferences.

Dr.Antony Selvadoss Thanamani is presently working as Professor and Head in the Dept of Computer Science, NGM College, India. He has published more than twenty papers in National/International journals and more than ten books. His areas of interest include E-Learning, Software Engineering, Data Mining, Networking and etc. He has about 20 years of teaching experience. He is guiding many research scholars and has published many papers in national and international conference and in many international journals