Design and Development of an Extensible Test Generation Tool based on the Eclipse Rich Client Platform

Angelo Gargantini[1] and Gordon Fraser[2]

[1] Università degli Studi di Bergamo, Italia
angelo.gargantini@unibg.it
[2] Saarland University – Computer Science
Saarbrücken, Germany
fraser@cs.uni-saarland.de

Abstract. In aspiration of automated software testing, a common task is the derivation of test cases from models. The wealth of different test criteria, model formalisms, and testing strategies makes reusability of such test generation tools a very challenging task. Leveraging the flexibility offered by the Eclipse Rich Client Platform, we present a new test generation tool that achieves reusability by abstracting from specific details of the test generation, and by matching these features with Eclipse extensions. The resulting tool allows the configuration of different backends for extracting tests from models, input languages, test strategies, and test criteria via plug-ins.

1 Introduction

Software testing remains the prevalent method to determine and to improve the quality of software. As exhaustive testing is impossible, coarser test criteria are used in practice. Because these criteria can only give an intuition of quality but never prove the absence of errors, researchers have come up with a great number of different criteria with different characteristics.

It is possible to interpret most of these criteria in a common framework that allows leveraging the power of modern model checking tools as vehicles for test generation: A test criterion is represented as a set of temporal logic formulas, for which the model checker can efficiently derive witnesses and counterexamples, serving as test cases. This offers a nice theoretical framework for test generation, but to instantiate this framework in practice requires a number of decisions: input language, test strategy, model checking tool, test criteria — a tool that commits to a particular choice of these decisions is likely to be unusable for many other practical applications.

Clearly, there is a need for an *extensible* tool for test generation that allows customization with respect to many different aspects. In this paper we present ExTGT (EXtensible Test Generation Tool), a test generation tool that can be customized in terms of plug-ins, thus allowing adaptation to many different practical settings. The implementation of ExTGT leverages from the possibilities

offered by the Eclipse Rich Client Platform. It offers a powerful environment which defines a set of extension points so building a family of test generation tools, each defining its own extensions as plug-ins for ExTGT. In the classical Eclipse style, ExTGT itself is a collection of extensions both for the Eclipse platform and for its own extension points.

This paper is organized as follows: Section 2 gives a general overview of model based testing based on model checking techniques. Section 3 contains the details of the implementation of the Eclipse based test generation tool, and shows how ExTGT has been extended. Finally, Section 4 concludes the paper and gives an outlook on how ExTGT will be further extended in the future.

2 Model based Test Generation by Model Checking

ExTGT is a model based testing tool: A test model describes the desired behavior, and test case generation means sampling execution paths of this test model. To perform this test generation, ExTGT uses model checking. A model checker is a formal verification tool which takes as input a model and a property, and then exhaustively verifies whether the property holds on the model or not. A nice feature of model checkers is their ability to derive counterexample and witness sequences – these sequences are essentially test cases.

While model checking is an industrially accepted technique in the hardware domain, it is also very useful in the software domain. Model checking can be used to automatically prove certain properties on programs, but this is not sufficient to replace software testing in general. The applicability of model checking is limited by the scalability of the chosen model checking techniques, and so verification of software is currently limited to small programs. Model checking as part of a software development cycle is often more practical when applied to more abstract artifacts such as models and formal specifications. However, proving correctness of such an artifact with regard to certain properties does not automatically prove correctness of the actual system. Therefore, model checking cannot replace software testing. This is even true if exhaustive verification of a program's source code is possible – the actual system depends on many additional factors such as the compiler, the hardware and software environment, other components, etc. Consequently software testing cannot be replaced by model checking.

Counterexample generation is one of the most useful aspects of model checkers in practice. From a software testing point of view, counterexamples essentially are test cases. Mapping counterexamples to automatically executable test cases is usually a straight forward task, although the exact details depend on the system under consideration. Callahan et al. [7] and Engels et al. [13] initially proposed to use model checkers to automatically generate test cases. Many different techniques to systematically derive test cases with model checkers have been proposed since then. A large body of work considers test case generation based on coverage criteria. Here, the idea is to represent each coverage item described by the criterion as a temporal logic property (*trap property* or *test predicate*), such that any counterexample to the property is a test case that covers the underlying

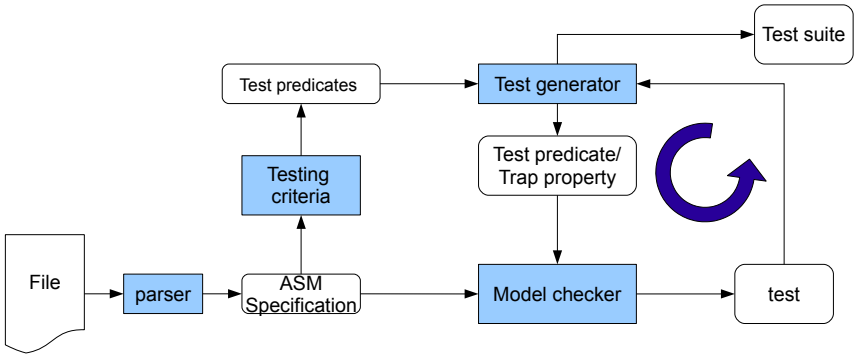

Fig. 1. Test Generation by Model Checking: A specification is parsed and serves as source for both, test predicates and model for the model checker. Test generation is performed by querying the model checker with the model and one test predicate at a time.

coverage item. Specification based structural coverage criteria are dominant in the literature [21, 22, 28], although some work has also been done on property based coverage criteria [17, 31, 34], and Hong et al. [25] describe control flow and data flow coverage criteria. Combinatorial coverage, where test cases for all pairs or tuples of input/state variables are required, is considered in [8, 26].

A similar approach is possible using mutation: In general, mutation describes a process where small changes are introduced in software artifacts in order to measure the quality of existing test sets, or in order to help generating new test cases that can detect these changes. Mutation can be applied to model checker specifications in order to automatically generate mutation adequate test suites. This has originally been proposed by [1, 3], and recently been refined by [27] and [20]. Mutation is sometimes also applied to models in order to force counterexample generation with respect to safety properties [2, 16].

Testing with software model checkers has been considered by [5], who use the model checker Blast [23] to create test cases from C code. Test cases can be generated with regard to predicates (i.e., safety properties), and locations in the source code. Consequently, it is possible to derive test cases for code-based coverage criteria. [33] use the Java PathFinder [32] model checker to derive test cases in a similar manner.

2.1 Test generation process

The test generation process employed by ExTGT is depicted in Figure 1: A specification file containing the Abstract State Machine [6] is read by the tool component *parser*. Starting from the ASM specification, according to the *testing*

criteria defined, the tool builds a set of test predicates. Note that automated test case generation requires formalization of the test objective (e.g., satisfaction of a coverage criterion), which can be seen as a set of test requirements (e.g., one test requirement for each coverable item). Each test requirement can be formalized as a temporal logical *test predicate*. Therefore, every testing criterion produces in practice a list of test predicates; this step can easily be automated. The *test generator* takes a test predicate at the time (following an ordering policy [15] or following user requests), builds the trap property, and call the model checker to get the counter example for that trap property. The counter example is converted to a test and given back to the test generator which extracts other useful information (for example the coverage of other test predicates). The test generation process by model checking is iterated until the desired test suite satisfies all feasible coverage goals.

Similarly, in fault based testing, test predicates are generated from the original conditions by applying mutation operators [20]. In this approach, the testing criteria defines a list of test predicates for *every* mutation operator.

3 Design and Development of an Extensible Test Generator Tool

Eclipse RCP allows developers to use the Eclipse platform to create flexible and extensible desktop applications. Eclipse itself is built upon a plug-in architecture and plug-ins are the smallest deployable and installable software components of Eclipse. This architecture allows the Eclipse applications to be extended by third parties. Eclipse RCP provides the same modular concept for stand-alone applications. An Eclipse RCP application can decide to use parts of the components provided by Eclipse, like editor, menus and so on. It is even possible to design headless Eclipse based applications which use only the Eclipse runtime.

Eclipse provides the concept of *extension points* and *extensions* to facilitate that functionality can be contributed to plug-ins by further plug-ins — plug-ins which define extension points open themselves up for other plug-ins. Such an extension point defines a contact how other plug-ins can contribute. The plug-in which defines the extension point is also responsible for evaluating the contributions. Therefore the plug-in which defines an extension point both defines the extension point and has some coding to evaluate contributions of other plug-ins.

A plug-in which defines an "extension" contributes to the defined "extension point"; the contribution can be done by any plug-in. Contributions can be code but also data contributions, e.g., help context. Extensions to an extension point are defined in the plug-ins via the file "plugin.xml" using XML.

The ExTGT core is itself a plug-in which provides several extensions to standard Eclipse components, like view, perspectives, menus, editors and so on. Moreover, ExTGT defines several extension points, presented in Section 3.1, which allow the addition of new functionalities, and it provides several extensions of the defined extension points as explained in Section 3.2. This setting is depicted in Fig. 2

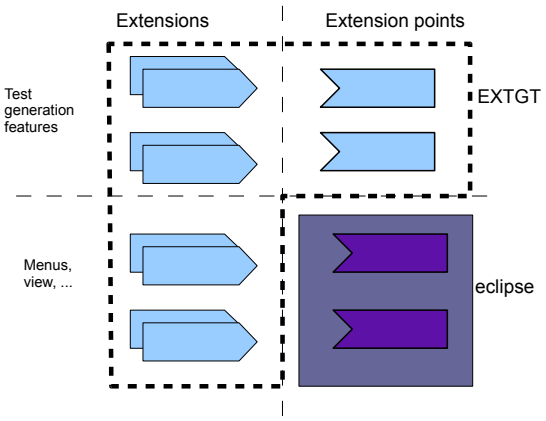

Fig. 2. ExTGT extensions and extension points: ExTGT defines its extension points based on common features of test generation tools. ExTGT is itself a plug-in to Eclipse and provides several extensions to standard Eclipse components such as views, perspectives, etc.

3.1 Extension points

ExTGT defines several extension points (see Figure 2), which are briefly explained in this section. In our approach each extension point has a reference abstract class or interface which is required to be extended or implemented by the extension of that extension point. The extension points are as follows:

extgt.asmSpecReader: This extension point allows to introduce several parsers for ASM specifications. Indeed, although the Abstract State Machine formalism is precisely defined in theory [6], several dialects exist for writing ASMs, and the designer can add the capability to read new formats by extending the class `AsmSpecReader` and introducing new parsers.

extgt.coverageBuilders: This extension point allows to define new coverage criteria. A coverage criterion is defined by the interface `AsmCoverageBuilder` which, given an ASM specification, builds a list of test predicates.

extgt.faultExpression: This extension point allows to introduce new mutation operators, which must extend the `FaultExpressionVisitor` class. Every extension must define a method which takes an expression e and returns the possible faulty implementations of e.

extgt.generatorMethod: New test generator methods can be introduced by extending the class `TestGeneratorMethod`. A `TestGeneratorMethod` is able to generate a test sequences starting from a test predicate by exploiting the model checking counter example feature.

3.2 Extensions

ExTGT provides a set of extensions for the extension points introduced in Section 3.1 (see Figure 2):

- The extension point `extgt.AsmSpecReader` has been extended by `AsmetaLLoader` in order to read Asmeta specifications as defined by Gargantini et al. [19], and by `AsmGoferLoader` to read AsmGofer [29] specifications.
- `extgt.coverageBuilders` has been extended by `BasicRuleVisitor`, `CompleteRuleVisitor`, `RuleUpdateVisitor`, and `MCDCCoverage` which compute the classical structural coverage as defined by Gargantini and Riccobene [18]. Combinatorial testing is introduced by the extension `PairwiseCovBuild` [9], while fault based testing is defined by the `PluggableFaultBasedCov` extension [20].
- `extgt.faultexpression` has been extended by all the faults defined by Gargantini [20], which are: [ENF] Expression negation fault (it consists in replacing a sub expression with its negation), [LNF] literal negation fault, [MLF] missing literal fault, [ST0/1] Stuck at 0 or at 1 fault, [ASF] Associative Shift fault, and [ORF] Operator Reference fault.
- `extgt.generatorMethod` is extended by several plug-ins, each implementing the test generation by a different model checker. ExTGT currently supports Spin [24], an explicit state model checker, the BDD and SAT based model checker SAL [11] (only for combinatorial testing), HySAT [14], a satisfiability checker for Boolean combinations of arithmetic constraints over real- and integer-valued variables which can also be used as a bounded model checker for hybrid (discrete-continuous) systems, and the Yices [12] SMT solver.

3.3 Architecture

ExTGT is built upon ATGT, a tool for test generation we have developed over the last years [4]. ATGT is compound of several packages as shown in Figure 3, and it comes in two variants: atgt_cli for a command line version, and atgt_swing, which offers a graphical user interface. Although ATGT was developed well before ExTGT, it already uses several projects, each defining a functionality. In terms of packages,ExTGT only needed a new package extgt_rcp, which defines the RCP application but which re-uses all the code already implemented by ATGT. Note that many graphical elements in ExTGT were reused from atgt_swing by using the Swing-SWT bridge.

3.4 Testing

Besides the usual JUnit tests for functional testing, to test ExTGT we use SWTBot [30]. SWTBot is an open-source Java based UI/functional testing tool for testing SWT and Eclipse/RCP based applications.

SWTBot provides several APIs that are simple to read and write. The APIs also hide the complexities involved with SWT and Eclipse. Furthermore, SWTBot provides its own set of assertions that are useful for SWT.

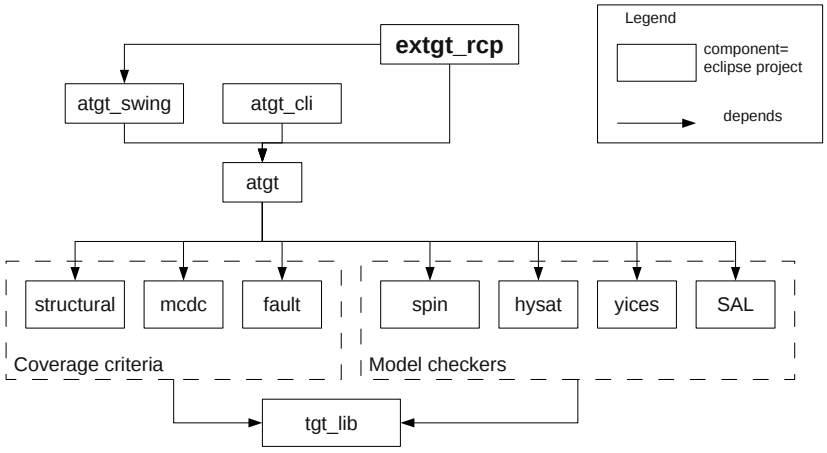

Fig. 3. ExTGT projects: ExTGT reuses many of the components of the ATGT code base [4]. These components mainly cover different model checkers and test criteria.

Listing 1. A snippet of an SWTBot test case as was used to test ExTGT.

```
// test the opening of a file
@Test
public void openFile() throws Exception {
        SWTBotMenu filemenu = bot.menu("File");
        SWTBotMenu openm = filemenu.menu("Open");
        openm.click();
        ...
}
```

The SWTBot tests were defined in a separate project, each in a different Java class. The methods to test the application consists in a sequence of commands simulating the use of the application. For example, Listing 1 illustrates a fragment of a test that opens a specification file:

3.5 ExTGT at work

A screenshot of ExTGT is presented in Figure 4. The user is not aware that ExTGT is an Eclipse-based application since it is not an Eclipse plug-in but a real stand alone application and has limited reuse of the classical Eclipse workbench elements.

The user loads the ASM specification in the tool. For instance, Listings 2 reports the specification of a Cruise Control system in AsmetaL.

Listing 2. AsmetaL Specification of a Cruise Control

```
// the cruise control module
asm cruiseControl
import StandardLibrary
signature:  //declare universes and functions
        enum domain CCMode  = {OFF| INACTIVE|CRUISE|OVERRIDE}
        enum domain CCLever  = {DEACTIVATE| ACTIVATE|RESUME}
        dynamic controlled  mode : CCMode
        dynamic monitored lever : CCLever
        dynamic monitored igOn : Boolean
        dynamic monitored engRun : Boolean
        dynamic monitored brake : Boolean
        dynamic monitored fast : Boolean

definitions:
// AXIOMS
axiom inv_ignition over engRun : (engRun implies igOn)
axiom inv_toofast over fast : (fast implies engRun)

//Rules
main rule r_CruiseControl =
if not igOn then  mode := OFF
else if  not engRun then mode:= INACTIVE
// igOn and engRun
else par
  if mode = OFF                  then mode := INACTIVE endif
  if mode = INACTIVE and not brake and not fast
             and lever = ACTIVATE then     mode := CRUISE
  endif
  if mode = CRUISE then
      if fast then mode := INACTIVE
      else if brake or lever = DEACTIVATE
                                then mode := OVERRIDE
      endif endif
  endif
  if mode = OVERRIDE and not fast and  not brake and
    (lever = ACTIVATE or lever = RESUME) then mode := CRUISE
  endif
  endpar
endif
endif
// initial state
default init s1:
   function mode = OFF
   function lever = DEACTIVATE
   function igOn = false
   function engRun= false
   function brake = false
   function  fast = false
```

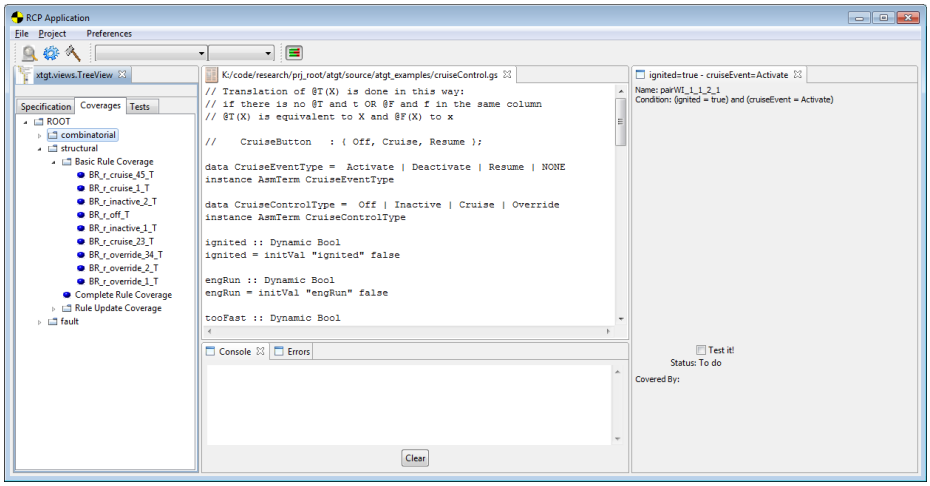

Fig. 4. Screenshot showing ExTGT in action: The coverage view shows the available test predicates, and ExTGT also shows the specification and details of the currently selected test predicate.

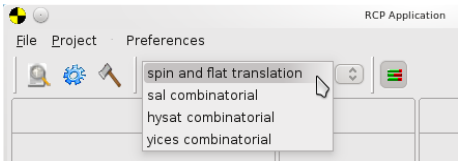

Fig. 5. Choosing a model checker plug-in in ExTGT: The user does not need to be aware of the extension points and extensions, but gets to see the effects in various options.

The user does not need to know which plug-ins are installed, but the application is aware of the extensions defined by its extension points. For example, the choice of the model checker is presented by the user by means of the simple pull-down menu shown in Figure 5.

In this case the application searches the list of the extensions defined as `extgt.generatorMethod` and builds the list of the available methods to be proposed to the user. After selecting a test generator method and the desired test predicates, the user can run the test generator of ExTGT to obtain the desired test cases.

4 Conclusions and Future Work

The Eclipse RCP framework has allowed us to use the Eclipse platform to create a flexible and extensible test generator tool, reusing some of the code we previously developed for ATGT. We plan to define new extensions, like the use of

further model checkers such as NuSMV [10] as test generator method. We plan also to define new extension points, for example the ordering policies in which test predicates are taken by the test generator. A major extension point we are planning to define is the specification notation to make ExTGT able to read not only ASM specifications, but also other formal notations like SCR or UML state machines.

Acknowledgments Angelo Fumagalli and Matteo Foiadelli developed the initial prototype of EXTGT. Laura Bottanelli developed the hysat module.

References

1. Paul Ammann and Paul E. Black. A Specification-Based Coverage Metric to Evaluate Test Sets. In *HASE '99: The 4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 239–248, Washington, DC, USA, 1999. IEEE Computer Society.
2. Paul Ammann, Wei Ding, and Daling Xu. Using a Model Checker to Test Safety Properties. In *Proceedings of the 7th International Conference on Engineering of Complex Computer Systems (ICECCS 2001)*, pages 212–221. IEEE, 2001.
3. Paul E. Ammann, Paul E. Black, and William 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, 1998.
4. ATGT Abstract State Machines test generation tool project. `http://cs.unibg.it/gargantini/software/atgt/`.
5. Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating Tests from Counterexamples. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04, Edinburgh)*, pages 326–335. IEEE Computer Society Press, 2004.
6. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
7. John Callahan, Francis Schneider, and Steve Easterbrook. Automated Software Testing Using Model-Checking. In *Proceedings 1996 SPIN Workshop*, August 1996. Also WVU Technical Report NASA-IVV-96-022.
8. Andrea Calvagna and Angelo Gargantini. A Logic-Based Approach to Combinatorial Testing with Constraints. In *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 66–83. Springer-Verlag, 2008.
9. Andrea Calvagna and Angelo Gargantini. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, 2010.
10. Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A New Symbolic Model Verifier. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 495–499, London, UK, 1999. Springer-Verlag.
11. Leonardo de Moura, Sam Owre, Harald Rueß, John Rushbyand N. Shankar, Maria Sorea, and Ashish Tiwari. Sal 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.
12. B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI Available at http://yices.csl.sri.com/tool-paper.pdf, 2006.

13. André Engels, Loe Feijs, and Sjouke Mauw. Test Generation for Intelligent Networks Using ModelChecking. In Ed Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 384–398, Enschede, the Netherlands, April 1997. Springer-Verlag.
14. Martin Fränzle and Christian Herde. Hysat: An efficient proof engine for bounded model checking of hybrid systems. *Form. Methods Syst. Des.*, 30(3):179–198, 2007.
15. Gordon Fraser, Angelo Gargantini, and Franz Wotawa. On the order of test goals in specification-based testing. *Journal of Logic and Algebraic Programming*, 78(6):472–490, July 2009.
16. Gordon Fraser and Franz Wotawa. Property Relevant Software Testing with Model-Checkers. *SIGSOFT Software Engineering Notes*, 31(6):1–10, 2006.
17. Gordon Fraser and Franz Wotawa. Complementary criteria for testing temporal logic properties. In Catherine Dubois, editor, *Proceedings of theThird International Conference on Tests And Proofs (TAP)*, volume 5668 of *Lecture Notes in Computer Science*, pages 58–73, Zurich, Switzerland, 2009. Springer.
18. A. Gargantini and E. Riccobene. Asm-based testing: Coverage criteria and automatic test sequence generation. *JUCS - Journal of Universal Computer Science*, 7(11):1050–1067, nov 2001.
19. A. Gargantini, E. Riccobene, and P. Scandurra. A metamodel-based language and a simulation engine for abstract state machines. *Journal of Universal Computer Science (JUCS)*, 14(12):1949–1983, 2008.
20. Angelo Gargantini. Using model checking to generate fault detecting tests. In *International Conference on Tests And Proofs (TAP), Zurich, Switzerland on 12-13 February 2007*, volume Lecture Notes in Computer Science (LNCS), pages 189–206, 2007.
21. Angelo Gargantini and Constance Heitmeyer. Using Model Checking to Generate Tests From Requirements Specifications. In *ESEC/FSE'99: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687, pages 146–162. Springer, 1999.
22. Gregoire Hamon, Leonardo de Moura, and John Rushby. Generating Efficient Test Sets with a Model Checker. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 261–270, 2004.
23. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software Verification with Blast. In *Model Checking Software: 10th International SPIN Workshop, Portland, OR, USA, May 9-10, 2003. Proceedings*, pages 235–239. Springer-Verlag, 2003.
24. Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.
25. Hyoung S. Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A Temporal Logic Based Theory of Test Coverage and Generation. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *Lecture Notes in Computer Science*, pages 151–161. Springer Verlag Gmbh, 2002.
26. D. Richard Kuhn and Vadim Okun. Pseudo-Exhaustive Testing for Software. In *30th Annual IEEE / NASA Software Engineering Workshop (SEW-30 2006), 25-28 April 2006, Loyola College Graduate Center, Columbia, MD, USA*, pages 153–158. IEEE Computer Society, 2006.

27. Vadim Okun, Paul E. Black, and Yaacov Yesha. Testing with Model Checker: Insuring Fault Visibility. In Nikos E. Mastorakis and Petr Ekel, editors, *Proceedings of 2002 WSEAS International Conference on System Science, Applied Mathematics & Computer Science, and Power Engineering Systems*, pages 1351–1356, 2003.
28. Sanjai Rayadurgam and Mats P. E. 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, Washington, DC, April 2001. IEEE Computer Society.
29. J. Schmid. AsmGofer. `http://www.tydo.de/AsmGofer`.
30. Swtbot - ui testing for swt and eclipse. `http://www.eclipse.org/swtbot/`.
31. Li Tan, Oleg Sokolsky, and Insup Lee. Specification-Based Testing with Linear Temporal Logic. In *Proceedings of IEEE International Conference on Information Reuse and Integration (IRI'04)*, pages 493–498, 2004.
32. Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model Checking Programs. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, pages 3–11, Washington, DC, USA, 2000. IEEE Computer Society.
33. Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test Input Generation with Java PathFinder. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, New York, NY, USA, 2004. ACM Press.
34. Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage Metrics for Requirements-Based Testing. In *ISSTA'06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 25–36, New York, NY, USA, 2006. ACM Press.