# Towards a Framework for Generating Tests to Satisfy Complex Code Coverage in Java Pathfinder *

Matt Staats
Department of Computer Science and Engineering
University of Minnesota
staats@cs.umn.edu

**Abstract**

We present work on a prototype tool based on the JavaPathfinder (JPF) model checker for automatically generating tests satisfying the MC/DC code coverage criterion. Using the Eclipse IDE, developers and testers can quickly instrument Java source code with JPF annotations covering all MC/DC coverage obligations, and JPF can then be used to automatically generate tests that satisfy these obligations. The prototype extension to JPF enables various tasks useful in automatic test generation to be performed, such as test suite reduction and execution of generated tests.

## 1 Introduction

When using a rigorous coverage criterion to generate tests, significant resources are often required to create tests that satisfy the criterion. Approaches for reducing the cost of generating test cases are thus of great value. One such approach is to automate the test case generation process by using a model checker to generate the tests. While several variations of this approach exist [2, 5, 10], the approach generally operates in two steps. First, the required coverage obligations are expressed as either states or sequences of states of the system for which we wish to generate tests. Second, for each state or sequence of states corresponding to a coverage obligation, the model checker is used to determine the reachability of the state or the existence of the sequence of states. If the model checker determines that no states representing a particular coverage obligation are reachable (a plausible occurrence for a complex coverage criterion), the tester has conclusive proof that said obligation is not coverable; otherwise, the model checker emits a trace that can be used as a test case. This approach is largely indifferent to the coverage criterion used – essentially the same approach can be used to generate tests for any coverage criterion that can be expressed in terms of states. Developers can reuse the state space exploration functionality provided by a model checker to quickly create test generation capability for a variety of coverage criteria.

In this paper, we describe the development of a prototype tool built on the Java Pathfinder (JPF) model checker that allows developers to generate and execute tests for the MC/DC coverage criterion [3]. The tool has been developed as two components: an Eclipse plugin allowing developers and testers to instrument Java source files with MC/DC coverage obligations and an extension to JPF providing the functionality needed to generate test cases from said instrumentation.

While the prototype is useful in its current state, it was developed primarily to explore the use of JPF for automatic test generation for reactive systems common to the safety critical domain. Our interest in using JPF over other model checkers (as used in our previous work [5, 8, 10]) stems from its structure; unlike many model checkers, JPF has been developed to be understandable and extensible. For this reason, we believe that JPF may be an excellent model checker for rapidly developing and empirically testing coverage criteria, and we view this prototype as the first step towards an extensible framework for the rapid development of automatic test generation functionality in JPF. This framework would be of use not only for developers and testers who want to build automatic test generation functionality for new and currently unsupported coverage criteria, but also for software researchers exploring the effectiveness of various coverage criteria.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 116-120

*This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, and the L-3 Titan Group.

## 2 Java Pathfinder

Java Pathfinder is a model checker for the Java language built on top of a custom Java Virtual Machine (JVM). Model checking is done via execution of Java bytecodes, an approach that allows different bytecode interpretations to be developed. This ability is used to support two major model checking modes in JPF: an explicit-state mode similar to SPIN [6], and a symbolic execution mode aptly named *Symbolic JPF* [1]. These modes differ primarily in how the state space is explored. During explicit-state execution, bytecode instructions are executed as in standard JVMs, such as the official Sun JVM. Nondeterminism can be either explicitly introduced by developers and testers as source code annotations in the program being analyzed or implicitly introduced when analyzing multi-threaded programs. JPF handles nondeterminism by branching the execution for each possible nondeterministic choice. Through this branching, the entire state space of the program being analyzed is explored. Multiple search algorithms are available and operate by influencing the order in which JPF explores branches (note that the default search is depth-first).

Symbolic JPF uses an extended interpretation of bytecodes to work with symbolic values, and contains functionality to create symbolic values in addition to concrete values. Operations over symbolic values are done algebraically in a straightforward manner. (The details of symbolic execution are omitted here; the interested reader is referred to [7] for specifics.) When Symbolic JPF encounters conditional branches incorporating symbolic values, it attempts to determine if the branch condition is satisfiable for both *true* and *false* possibilities using off-the-shelf decision procedures. The execution paths of any satisfiable possibilities are then explored, thus branching the search. Note that multi-threading, source code annotations (nondeterministic and otherwise) and operations over concrete values are handled as in explicit-state execution.

A substantial number of helper functions and classes have been developed for JPF. These allow developers and testers to annotate code (including the nondeterministic choice annotations mentioned previously), and allow extensions to modify and monitor the execution of JPF. One of the more powerful pieces of functionality is the ability to register Java listeners for various JPF events, such as execution of a bytecode instruction or the processing of a non-deterministic choice. This functionality allows extensions to access essentially the same information available internally to JPF. As we will see later, both the ability to annotate code and the ability to monitor JPF's execution greatly aides in the development of test generation functionality.

### 2.1 Existing Test Generation Functionality

JPF contains some preexisting functionality useful for generating tests. When generating tests exclusively using concrete execution mode, source code annotations are generally used. Nondeterministic choice annotations, such as *Verify.getBoolean()* and *Verify.getInt()*, provide a simple way for testers to simulate receiving inputs from the user and/or environment. Generating tests is accomplished through *storeTraceIf(condition, fileName)* statements. These statements, if executed when *condition* evaluates to *true*, output to *fileName* a list of values chosen for each nondeterministic choice. In principle, this list, or *trace*, can be examined to determine an execution path in which *condition* is true at the point *storeTraceIf* produced the trace. Once the execution path is identified, constructing a test should be relatively easy. In practice, however, these traces are only marginally useful, since (1) there exists no method of running or replaying the trace and (2) the trace is output in a format that is not readily understandable.

Using Symbolic JPF to generate tests can be done by assigning symbolic, rather than concrete, values to one or more variables. When Symbolic JPF is used to analyze the program, *path conditions* that algebraically represent the state of variables are produced. By solving these path conditions, concrete values satisfying the path conditions can be found. Thus, Symbolic JPF can be used to generate tests

by finding a path of interest (e.g., a path leading to a coverage obligation) and solving the resulting path conditions. In fact, Symbolic JPF includes functionality for finding all paths within a method (up to some maximum length) and generating a test for each path, thus yielding path coverage over the method. However, while Symbolic JPF can be very effective for generating tests, no method of assigning symbolic values within a loop exists, thus precluding test generation for reactive systems (which are constructed as essentially large infinite loops).

# 3 Prototype MC/DC Test Generation Tool

We chose to develop the prototype to generate tests for the MC/DC coverage criterion because (1) it is a fairly complex coverage criterion and thus presents at least somewhat of a challenge to implement and (2) it is used in critical systems software such as avionics software. We do not describe the MC/DC coverage criterion in this paper; interested readers are directed to [3].

The prototype addresses some of the limitations present in JPF's existing test generation capability and introduces new functionality. The tool consists of two components: an extension to JPF that improves code annotations and adds support for a variety of common test generation tasks, and a plugin for the Eclipse IDE [4] to instrument Java source code with MC/DC test obligations.

## 3.1 JPF Extension

We developed the extension to JPF with two goals in mind. First, improve the usability and understandability of tests generated via code annotations. Second, provide functionality allowing tasks common in automatic testing such as test suite reduction and code coverage measurement to be performed directly in JPF.

Improving the code annotations was a fairly straightforward task. As we explained in Section 2, the *Verify* class can be used to generate traces via code annotations, but the traces generated are not practical for use as tests. We therefore re-implemented the code annotations to allow more useful traces to be generated. This resulted in a *Debug* class containing three key improvements to the code annotations.

First, we created choice annotations that produce symbolic values (i.e., we created the symbolic equivalents of *Verify.getInt()*) thus allowing developers and testers to explicitly assign symbolic values at any point in their source code. These annotations can be safely used within loops, thus bringing the benefits of symbolic execution to reactive systems.

Second, we extended choice annotations to allow them to be "tagged" with a string by developers and testers. Provided the developer or tester chooses tags which are meaningful (e.g., "*className.variableName*"), traces produced are far more understandable. Furthermore, these tags offer a simple way to associate information useful to JPF with a choice annotation.

Finally, we added the ability to execute traces generated by the *Debug* class. When starting JPF, developers or testers can specify a trace previously generated by *Debug*. During JPF's startup, the specified trace is parsed by *Debug*, which then creates a queue of values for each tag in the trace. After JPF begins running, when a choice annotation is executed, *Debug* checks the tag associated with the annotation and pops a value from the appropriate queue. If the appropriate queue is empty the standard choice annotation logic is used. Executing a trace thus results in JPF following the execution path implied by the trace until reaching the end of said path, after which JPF's usual operation resumes.

These code annotation improvements allow useful, executable tests to be generated using JPF. Given some method to instrument code for a coverage metric (which we provide as an Eclipse plugin, described next) they provide a viable method of generating tests for Java code. However, when using automated test generation, there are several tasks beyond test generation developers may wish to perform, including the ability to measure code coverage, reduce test suite size while maintaining code coverage, and execute

entire test suites automatically (e.g., when performing regression testing). Rather than force developers and testers to develop ad-hoc methods of accomplishing these tasks, we provide functionality for these tasks in the form of JPF command line options.

We implemented much of this additional support by mimicking JPF's extensive use of Java listeners; specifically, we implemented a generic *DebugListener* class that can be registered with the *Debug* class to listen for certain test generation-related events during JPF's execution. Additionally, we created a test harness to execute tests in a test suite sequentially. By combining the test harness with listeners tracking *storeTraceIf* events, we perform test suite reduction and code coverage measurement for the MC/DC test coverage criterion. Note that while this functionality has been developed for generating and reducing test suites for the MC/DC coverage criterion, it could be easily adapted to other coverage criteria; most of the functionality exists in generic, extensible classes (in fact, it is only the tagging conventions used by the MC/DC Eclipse plugin that make this functionality coverage specific).

For complex coverage criteria such as MC/DC, even modestly sized systems may require thousand of coverage obligations to be generated, converted into the appropriate *Debug.storeTraceIf* statements, and inserted into the code at the appropriate points. Manually performing this is both tedious and error prone; thus while the functionality outlined above provides an approach for automatically generating tests, it is of limited use without a method of automatically instrumenting the source code to generate tests.

### 3.2 MC/DC Instrumentation Eclipse Plugin

The Eclipse plugin automates the tedious work of instrumenting a Java source file with the annotations needed to generate tests satisfying the MC/DC coverage criterion. Using the plugin is simple: when a user right clicks on a Java source code file in Eclipse, the options "Instrument to MC/DC Coverage" and "Remove MC/DC Instrumentation" are presented (in addition to the many other options Eclipse offers). When instrumenting, *Debug.storeTraceIf(obligation, fileName, tag)* statements are inserted into the source file, one for each MC/DC obligation. Each statement is inserted prior to the condition from which the *obligation* is generated from. Each *tag* contains a unique number for its corresponding obligation as well as the total number of obligations generated (used for calculating coverage). Given these obligations and the functionality described above, JPF can be used to (1) generate test cases meeting these obligations, (2) measure the MC/DC coverage achieved by an arbitrary set of test cases and (3) reduce a set of test cases while providing the same level of MC/DC coverage.

We have implemented this using Eclipse's Java abstract syntax tree. When instrumenting a Java source file, the file is parsed into an abstract syntax tree and each node in the tree is visited. When a condition node is encountered, the MC/DC obligations corresponding to the condition are generated using the method described in [10] and the appropriate *Debug.storeTraceIf(obligations, fileName, tag)* statements are inserted. Removal of the instrumentation is performed using a similar search.

## 4 Future Work and Conclusion

We view this prototype as the first step towards a framework for rapidly implementing automatic test generation capability for arbitrary coverage criteria in JPF. The key to this proposed framework is an approach (or combination of approaches) for quickly implementing automatic test generation for most coverage criteria using JPF. Though we have not conclusively identified a suitable approach, the development of our prototype has yielded several possibilities, including an Eclipse-based JPF code annotation library and a JPF extension for generating and monitoring code obligations during JPF's execution.

The development and use of a code annotation library seems well suited for structural coverage criteria. Many structural coverage criteria are designed to generate coverage obligations for specific

constructs in the source code, and thus the approach used to annotate for one coverage criterion could be adopted for many other coverage criteria. For example, by modifying what obligations are generated for each condition, the MC/DC Eclipse plugin could be adapted to generate code annotations for branch or decision coverage.

The ability to generate and monitor code obligations during JPF's execution is appealing from an end user perspective, as it frees users from requiring anything other than JPF. However, note that while most structural coverage criteria are expressed in terms of source code constructs, JPF operates exclusively over bytecodes. This complicates attempts to monitor source code constructs, as a single construct (say, a branch) may correspond to multiple bytecodes. Determining if a branch evaluates to true or false thus requires understanding how the path taken through the bytecodes corresponds to the path taken through the source code.

As mentioned in the introduction, we are interested in using JPF to automatically generate tests for testing safety critical systems. In particular, we are interested in working with our collaborators at NASA Ames to explore potential coverage criteria for Java code generated from Simulink using translation tools developed at Vanderbilt University [9]. To perform this exploration, we plan to use our proposed framework to quickly develop the automatic test generation capability for potential coverage criteria. Using this automatic test generation capability, we plan to then empirically evaluate said criteria using realistic systems developed at NASA.

## 5 Acknowledgements

We would like to thank Corina Pasareanu of NASA Ames for her comments on this paper as well as her significant contributions towards the prototype. We would also like to thank NASA Ames, and Mike Lowrey in particular, for their support in developing the prototype as well as their suggestions for future work.

## References

[1] S. Anand, C.S. Pasareanu, and W. Visser. JPF-SE: A Symbolic Execution Extension to Java PathFinder. *LECTURE NOTES IN COMPUTER SCIENCE*, 4424:134, 2007.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133. ACM New York, NY, USA, 2002.

[3] J. J. Chilenski and S. P. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*, pages 193–200, September 1994.

[4] E. Foundation. Eclipse IDE, 2006.

[5] Mats P.E Heimdahl, S. Rayadurgam, Willem Visser, George Devaraj, and Jimin Gao. Auto-generating test sequences using model checkers: A case study. In *3rd International Worshop on Formal Approaches to Testing of Software (FATES 2003)*, 2003.

[6] Gerald J Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[7] S. Khurshid, C.S. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 553–568, 2003.

[8] A. Rajan, M.W. Whalen, and M.P.E. Heimdahl. The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage. In *Proceedings of the 30th International Conference on Software engineering*, pages 161–170. ACM New York, NY, USA, 2008.

[9] J. Sztipanovits and G. Karsai. Generative Programming for Embedded Systems. In *International Conference on Principles and Practice of Declarative Programming*, volume 6, pages 180–180. Springer, 2002.

[10] M.W Whalen, A. Rajan, and M.P.E. Heimdahl. Coverage metrics for requirements-based testing. In *Proceedings of International Symposium on Software Testing and Analysis*, July 2006.