jMoped: A Test Environment for Java Programs*
(Tool Paper)

Dejvuth Suwimonteerabuth, Felix Berger, Stefan Schwoon, and Javier Esparza

Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany**

1 Introduction

We present jMoped [1], a test environment for Java programs. Given a Java method, jMoped can simulate its execution for all possible arguments within a finite range and generate coverage information for these executions. Moreover, it checks for some common Java errors, i.e. assertion violations, null pointer exceptions, and array bound violations. When an error is found, jMoped finds out the arguments that lead to the error. A JUnit [2] test case can also be automatically generated for further testing.

Initially, jMoped was developed as a text-based translator from Java bytecode into symbolic pushdown systems (SPDS). Technical details about the translation process can be found in [3]. Since then, we have extended the tool with two goals:

- *Harnessing the model-checking technique to support testing.* Model checking can symbolically test many inputs at the same time, is useful for finding boundary cases, and can provide coverage metrics.
- *Giving more control to the user.* The tool must allow users to inspect the intermediate results and to interrupt and refine the analysis at any time. Partial results should also be useful for further analyses.

The resulting tool has been developed as a plug-in for Eclipse [4], which is again called jMoped. It now consists of a graphical user interface, the translator, and Moped [5] at the back-end. Moreover, the translator itself has been improved in many aspects. It supports not only almost all fundamental features, e.g. assignment, method call, and recursion, but is also able to handle inheritance, abstraction, and polymorphism. On the other hand, it still fails to translate negative numbers, floats, and multi-threading programs.

2 Testing and Model Checking

Traditionally, testing and model checking are seen as distinct methodologies; testing can detect bugs but not prove their absence, and model checking seeks to establish the absence of bugs, possibly at the cost of taking very long to complete (or not finishing at all). Recently, several efforts have been made at

* Partially supported by the DFG-Project "Algorithms for Software Model Checking".
** This work was done while the authors worked at the University of Stuttgart.

W. Damm and H. Hermanns (Eds.): CAV 2007, LNCS 4590, pp. 164–167, 2007.

cross-fertilising between these two areas. Our tool falls into this line of work in the sense that we use a model checker to support the task of testing a program.

Internally, the tool translates a Java program into an SPDS, preserving the control flow of the program, but modelling only a finite amount of data. The size of variables and of the heap are bounded by user-defined (artificial) ranges. Thus, the model-checking procedure can be thought of as an extended symbolic testing procedure, which is still incomplete (because only runs within the given bounds are considered); however, once the bounds are established, the model checker will perform exact checks on *all* executions within these bounds.

We contend that this approach can complement traditional testing methods for two reasons: First, model checkers using compact data structures (such as BDDs) can simulate many executions at the same time, which can be more efficient than exhaustive testing. For example, our tool can test a Quicksort implementation for 60 array elements if each element has only one bit, whereas exhaustive testing for these parameters is infeasible. Secondly, model checking is suitable for finding boundary cases, i.e. inputs with special properties that are easily forgotten during testing, but are prone to cause bugs. E.g., two boundary cases for a sorting procedure would be an array where all elements are the same, or an array that is already sorted. Even relatively small bounds on the inputs are likely to contain many interesting boundary cases, and the model checker will test all of them (and find the faulty ones). Thus, the approach can greatly enhance the confidence in the correctness of a program, without strictly guaranteeing it.

The results of a model-checking procedure can support testing in other ways, too. The quality of a set of test cases is usually measured by so-called coverage metrics, e.g., counting how many lines of code were exercised by the test cases. We observe that such metrics can also be obtained by running a model checker on a set of inputs and checking which lines were found to be reachable. In jMoped, the user can observe the progress of these metrics while the checker is running. Moreover, the user may stop the checker at any time (e.g., if the attained level of coverage is deemed satisfactory), or ask it to specifically search for inputs that can reach a certain target in the program. Moreover, if the checker finds that bugs are caused by certain inputs, those inputs can be saved in a library of JUnit test cases, where they may be useful for future test runs.

3 Working with jMoped

jMoped consists of three parts: a graphical user interface, a translator from Java bytecode into SPDS, and an SPDS model checker. The translator and the checker are available as stand-alone tools, and the checker is capable of handling programs with thousands of lines, provided that the data complexity is low as is the case, for instance, with device drivers [5]. However, the graphical interface was developed for unit testing, with smaller, more data-intensive programs in mind. The interface is also described in more detail in [6].

The graphical interface takes the form of an Eclipse plug-in. Figure 1 shows an example when running with a Quicksort implementation taken from [7]. The

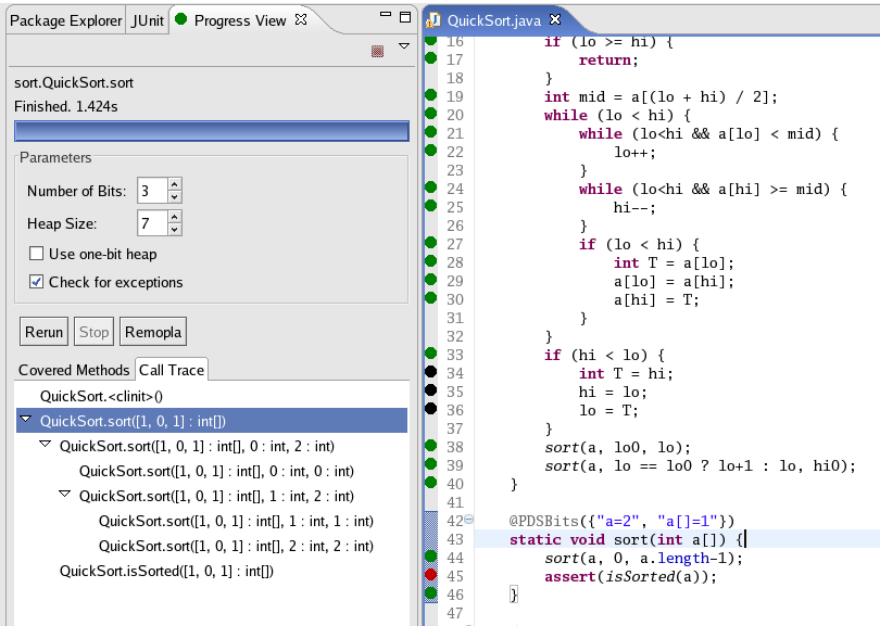

Fig. 1. A view of the plug-in

left-hand side is the plug-in interface, while the right-hand side shows parts of the code and the analysis results. Users select a method from which the analysis should start. In the example, the method `sort` starting at line 43 was chosen.

jMoped has two modes of operation. In the standard mode, jMoped exhaustively explores the program for all inputs within the bounds provided by the user. This is done in two steps. First, the program (which reads input from its user) is transformed into another program that nondeterministically generates an input. Then, the checker exhaustively explores all behaviours of the transformed program. In the second mode of operation, jMoped works backwards. Given a postcondition, jMoped computes the set of all states (within the given bounds) from which the states of the postcondition can be reached.

During the analysis, jMoped graphically displays its progress. First, black markers are placed in front of all statements. While the checker is running, the parts of the state space found to be reachable are mapped back to the Java program, and the appearance of the corresponding markers is changed. When a black marker turns green, it means that the corresponding Java statement is reachable from *some* argument values. A red marker means that an assertion statement has been violated by some argument values. Other markers indicate null pointer exceptions, array bound violations, and heap overflows (see below).

After the analysis, users can either create a *call trace* or a JUnit test case that reaches a given statement or violates some assertion. An example of the call trace can be seen in lower left part of Figure 1, where the assertion violation occured when the method `sort` was called with the array `[1,0,1]`.

In a typical scenario, a user will want to achieve 100% coverage, i.e. the checker should test a set of inputs such that every statement is exercised at least once. The idea for achieving this is to combine the two modes of operation. First, one uses the standard mode to cover as many instructions as possible. Say all but three instructions were covered. Then, in a second phase, one applies three backward searches starting from these three particular instructions. Since these are specific searches with the "difficult" instructions as goal, the hope is that they have better success chances than the "blind" forward search.

For performance reasons, the user starts the checker in standard mode with small values for the parameters, in the hope of achieving a large coverage degree quickly. However, choosing small parameters may cause some (normally reachable) statements to be considered unreachable. (E.g., the body of an if-statement guarded by the condition `x >= 8` would be unreachable with a specified bit size of less than 4.) Backward search can then be used to instruct the model checker to search for inputs that reach the remaining statements.

There are two important parameters to jMoped: the number of bits and the heap size. These determine the bounds for the inputs and executions that are to be tested. The number of bits restricts the range of every number that appears in a program, including constants, integer variables, and the lengths of arrays or strings. The heap size directly affects the number of objects that can be instantiated. jMoped simulates the Java heap when manipulating objects. For example, when an object is created, it occupies a part of the heap whose size depends on the size of the object. Note that these two parameters depend on each other, i.e. the heap size can be at most two to the number of bits minus one. For instance, the analysis in Figure 1 was performed with 3 bits and heap size 7. It is also possible to specify how many bits to use for individual variables, parameters, or fields. The annotation at line 42 of Figure 1 means that the length of array `a` has two bits, and each of its elements has one bit.

References

1. jMoped: A test environment for Java programs, http://www.fmi.uni-stuttgart.de/szs/ `tools/moped/jmoped/`
2. JUnit: Testing resources for extreme programming, `http://www.junit.org/`
3. Suwimonteerabuth, D., Schwoon, S., Esparza, J.: jMoped: A Java bytecode checker based on Moped (Tool paper). In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 541–545. Springer, Heidelberg (2005)
4. Eclipse: An open development platform, `http://www.eclipse.org`
5. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, Technische Universität München (2002)
6. Berger, F.: A test and verification environment for Java programs. Master's thesis, University of Stuttgart (2007)
7. ParForCE Project Workshop: Performance comparison between Prolog and Java, http://www.clip.dia.fi.upm.es/Projects/ParForce/Final_review/slides/i ntro/ `node4.html`