jMoped: A Test Environment for Java programs

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

Institut für Formale Methoden der Informatik, Universität Stuttgart,
Universitätsstr. 38, 70569 Stuttgart, Germany

1 Introduction

We present jMoped [1], a test environment for Java programs. Given a Java method, jMoped can simulate the execution of the program 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.

jMoped was first presented in [3] as a text-based translator from Java bytecode into symbolic pushdown systems (SPDS). 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 is seen as an incomplete method that can detect the presence of bugs but not prove their absence. The choice and design of test cases has been the subject of extensive research. In contrast, model checking is traditionally seen as a method that seeks to establish the correctness of a program, i.e. the absence of bugs, possibly at the cost of taking very long to complete (or not finishing at all). In recent years, there has been some cross-fertilising between these two areas, e.g. [6, 7]. Our tool differs from this line of work in the sense that a model checker is used to support the task of testing a program.

Internally, the tool translates a Java program into an SPDS. The procedural behaviour of a program under consideration is thus preserved (including recursion); however, in order to retain decidability, only a bounded amount of data can be modelled. Therefore, one needs to introduce (artificial) ranges on the size of variables and bound the size of the heap. Seen this way, the model-checking procedure can be thought of as an extended symbolic testing procedure, which is still incomplete (because the bounds prevent the checker from considering all executions); 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 tradition 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. 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. In the context of our tool, even relatively small bounds on the inputs are likely to contain most or even all 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 be useful for testing purposes in other ways, too. In testing, the quality of a set of test cases is measured by so-called coverage metrics. E.g., one well-known metric is to count how many lines of code were actually exercised when running the test cases (chances being that bugs may hide in untested lines of codes). We observe that such metrics can also be obtained by running a model checker on a set of inputs and analysing which parts of the state space were found to be reachable by the checker. In jMoped, the progress of the model checker (and the corresponding coverage metrics) can be observed by the user 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 the checker to specifically find 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. While the translator and the checker are available as stand-alone tools, they are more comfortably accessed through the graphical interface. This interface takes the form of an Eclipse plug-in [8].

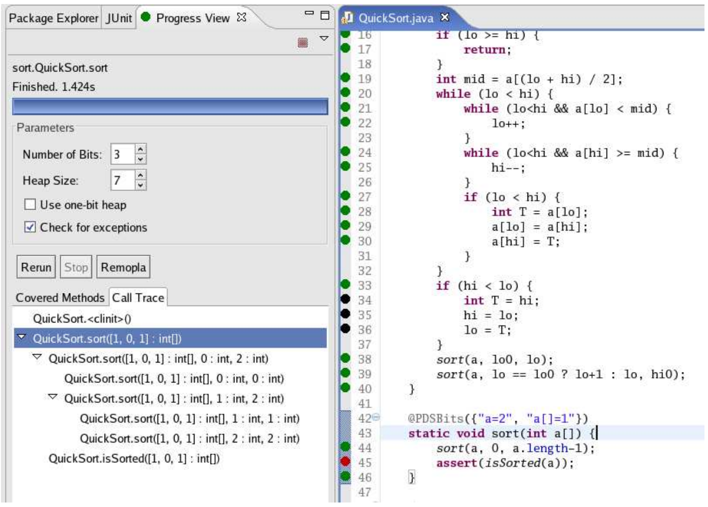

Fig. 1. A view of the plug-in.

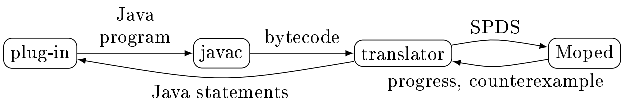

Fig. 2. Overview of the architecture.

Users select a method from which the analysis should start. Figure 1 shows an example of the plug-in when running with a Quicksort implementation taken from a website [9]. The left-hand side is the interface of the plug-in, while the right-hand side shows parts of the code and the result of the analysis. The method `sort` starting at line 43 was selected as the starting method.

When starting the analysis, jMoped generates a new method which calls the starting method with non-deterministic argument values within the ranges. Then, it statically analyzes which methods can be reached from the starting method, and puts black markers in front of all statements within these methods. After that, Moped is called to solve the reachability analysis of the underlying SPDS. While the checker is running, its progress is displayed graphically: the parts of the state space found to be reachable are mapped back to the Java program, which changes the appearance of markers. 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). Figure 2 illustrates the interplay between the components of the tool.

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]`.

There are two important parameters to jMoped, i.e. 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. This includes 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 must correspond to each other, namely the heap size can be at most two to the number of bits minus one. The analysis stops if a number is exceeded, and reports the error. For instance, the analysis in Figure 1 was performed with 3 bits and the 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 element has one bit.

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. For performance reasons, the user would start the checker 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.) Thus, if the model checker achieves less than 100% coverage, the user may try to increase the parameters and specifically instruct jMoped to search for inputs that can reach the statements that have not yet been covered. (Technically, this is done by backwards reachability analysis from a given statement.)

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. In: Proc. TACAS. LNCS 3440 (2005) 541–545 Tool paper.
4. Eclipse: An open development platform (http://www.eclipse.org)
5. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, Technische Universität München (2002)
6. Godefroid, P., Klarlund, N.: Software model checking: Searching for computations in the abstract or the concrete. In: Proc. of IFM. (2005) 20–32
7. Gulavani, B., Henzinger, T., Kannan, Y., Nori, A., Rajamani, S.: SYNERGY: A new algorithm for property checking. In: Proc. of SIGSOFT FSE. (2006) 117–127
8. Berger, F.: A test and verification environment for Java programs. Master's thesis, University of Stuttgart (2007)
9. ParForCE Project Workshop: Performance comparison between Prolog and Java (http://www.clip.dia.fi.upm.es/Projects/ParForce/Final_review/slides/intro/node4.html)