jMoped: A Test Environment for Java Programs

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

Technische Universität München, Germany

http://www7.in.tum.de/tools/jmoped/

Introduction

jMoped supports unit testing of Java programs using model-checking techniques. Given a Java method and a (finite) range of inputs, it performs a reachability analysis to check the program for these inputs. Highlights include:

- Symbolic testing: uses a BDD-based model checker for testing a large set of inputs.
- Generates *coverage* information from model-checking results.
- Tests for common Java errors (assertion violations, null-pointer exceptions, etc).
- Eclipse plug-in for browsing Java files, controlling the model checker, and viewing coverage information.
- Generates JUnit test cases for faulty inputs.

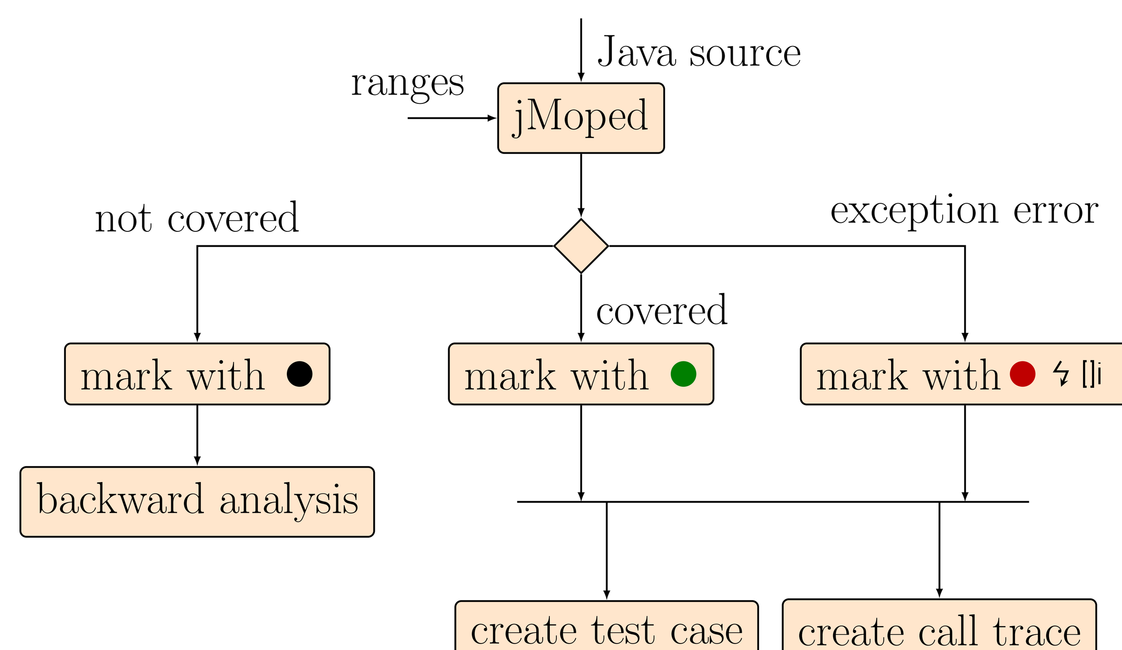

Figure 1: Overview of jMoped.

jMoped supports almost all fundamental features of Java, e.g. assignment, method call, recursion, exception, inheritance, abstraction, and polymorphism. On the other hand, it does not handle negative numbers, floats, and multi-threading programs.

```
20     void m(int i) {
21         if (i > 3) m(i-1);
22         assert(i < 3);
23         A x = new B(); assert(x.m() >= i);
24         x = new C(); assert(x.m() >= i+1);
25         try { throw new IllegalStateException(); }
26         catch (NumberFormatException e) { assert (false); }
27         catch (Exception e) {
28             if (i >= 2) { int a[] = new int[i]; a[i] = 0; }
29             else if (i >= 1) { int[] a = new int[4]; }
30             else { x = null; assert(x.m() == 1); }
31         }
32     }
```

Figure 2: A small Java code demonstrating jMoped features.

jMoped works smoothly with small programs, which could be numerically intensive, have many boundary cases, and make use of many features of the language. For larger programs, where analysing the entire state space is infeasible, jMoped offers an option to abstract some parts of the code or even the whole Java library.

Background

The tool consists of three parts: a graphical user interface (Eclipse plug-in), a translator, and a model-checker at the back-end.

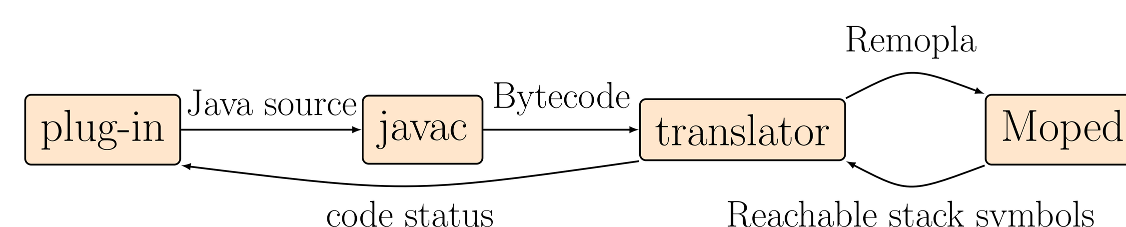

Figure 3: Overview of the architecture.

The model checker at the back-end is Moped, a model checker for symbolic pushdown systems (SPDS). *Remopla* is the language of Moped, which is essentially a shorthand for describing SPDS.

```
module int fac(int n(5)) {
  int m(5);
  if
  ::  (n==0 || n==1) -> return 1;
  ::  else -> m = fac(n-1); return n*m;
  fi;
}
```

Figure 4: A factorial program written in Remopla language.

The translator translates Java bytecodes into Remopla. Usually, one bytecode instruction is translated to one Remopla statement. The translation idea is summarized in the following table.

Bytecode	Remopla
Stack of frames	SPDS stack
(Bounded) Operand stack	Local variable
Local variable	Local variable
Static field	Global variable
Object manipulation	Heap simulation

The heap is simulated to handle objects. The simulation is achieved via a global array and a pointer:

- Every time a new object is created, it occupies some parts of the array.
- The pointer is always updated to the next available block of the array, which is determined by the size of the objects.
- E.g., `int[] a = new int[3];` ptr → a 3 ptr.

Also, every class is assigned a unique *id*:

- Used mainly for supporting polymorphism when there is a need to differentate types of objects that are stored in the array.
- Virtual fields of an object are kept in the heap, thus they determine the size of the object in the array.
- E.g., `class C { int v; }` and `C a = new C();` ptr → a id ptr.

Working with jMoped

Figure 5 shows an example when running with a Quicksort implementation found on the web. To start jMoped, users select a method from which the analysis should start. Here, the method `sort` starting at line 43 was chosen.

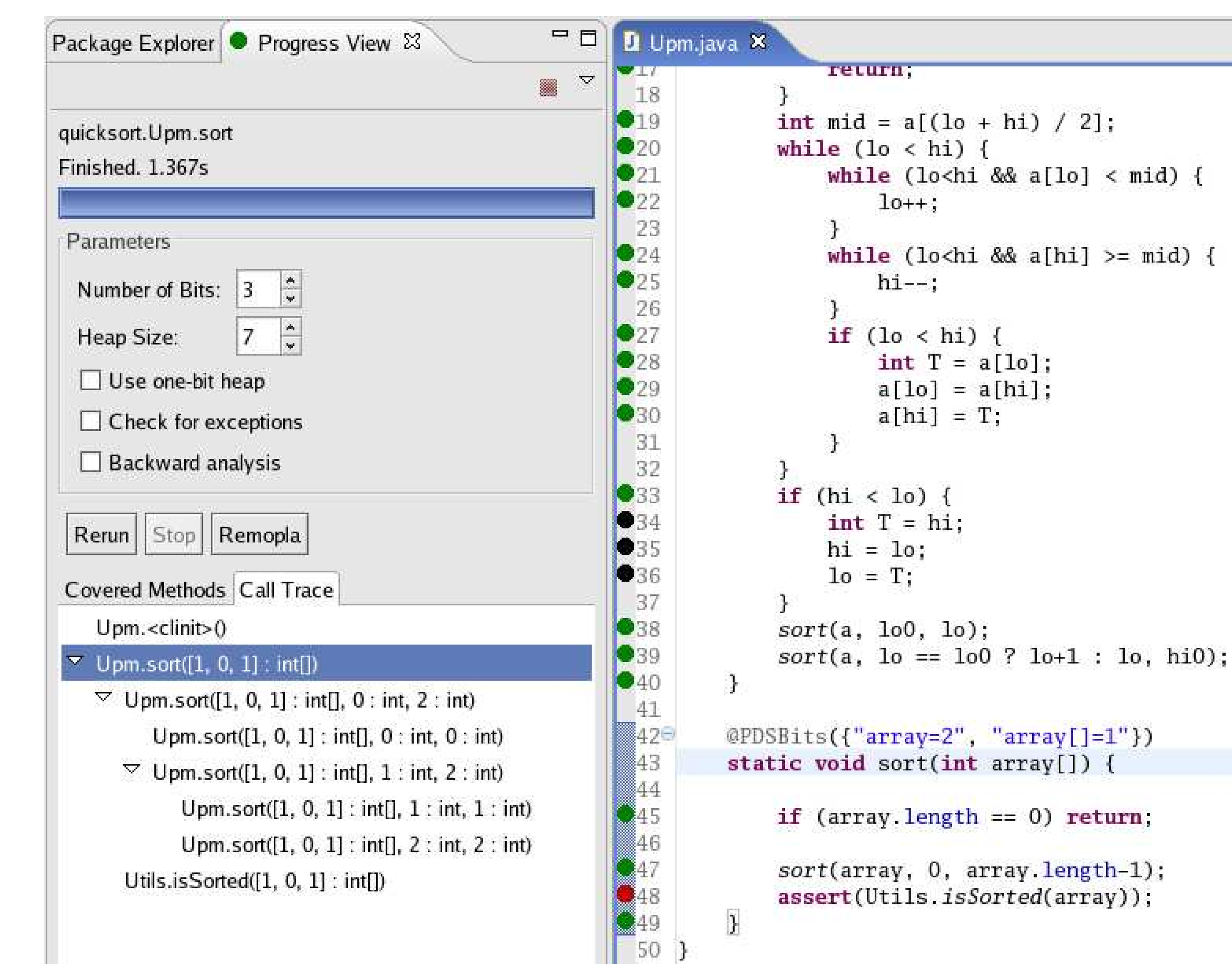

Figure 5: A view of the plug-in. The left-hand side is the plug-in interface, while the right-hand side shows parts of the code and the analysis results.

jMoped has two modes of operation.

(i) 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 inputs from its user) is transformed into another program that nondeterministically generates an input. Then, the checker exhaustively explores all behaviours of the transformed program.

(ii) Performs backward analysis starting from an instruction.

jMoped graphically displays its progress with the following markers.

● not covered	● covered
● assertion error	ϟ null-pointer exception
array bound violation	✕ heap overflow

After the analysis, users can either create a *call trace* or a JUnit *test case* that reaches a given statement or violates some assertion.

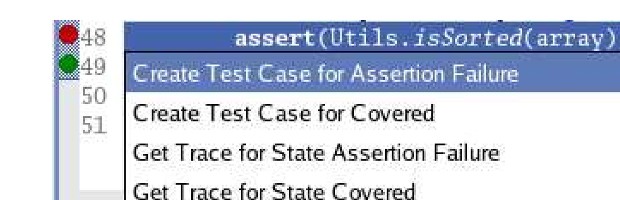

Figure 6: Options available when the red marker is clicked.