# jMoped: A Java Bytecode Checker Based on Moped*

Dejvuth Suwimonteerabuth, Stefan Schwoon, and Javier Esparza

Institut für Formale Methoden der Informatik, Universität Stuttgart
{suwimodh,schwoosn,esparza}@informatik.uni-stuttgart.de

**Abstract.** We present a tool for finding errors in Java programs that translates Java bytecodes into symbolic pushdown systems, which are then checked by the Moped tool [1].

## 1 Introduction

We present jMoped, a checker for (a large class of) Java programs. jMoped consists of a translator that transforms Java bytecode into a *symbolic pushdown system* (SPDS), which is then checked by the Moped tool [1]. The translator, described in more detail in [2], supports a wide range of Java programming features including arithmetic operations, control statements, method calls, recursion, arrays, object manipulations, inheritance, and exception handling. On the other hand, its current implementation does not support float, double, or string variables, dynamic arrays, non-static global arrays, dynamic method binding, calling a method of an interface, implicit exceptions, and multi-threading. Moreover, every instance of a class must be initialized with a separate `new` statement.

The functionality of jMoped is very simple. The user writes a Java class satisfying the constraints above (like the one on the left of Figure 1), and adds either a method `error()`, which is executed if some invariant is violated, or a method `ok()` signaling termination. For instance, in order to check if the variable $x$ may become zero at a certain program point, the user adds a line `if x = 0 then error()`. In the example of Figure 1, calling "`jmoped -e -`$n$ `LinkedList`", where $n$ is a number whose precise meaning is explained later, we obtain the answer that no execution of `error()` was found. However, if one changes the call in `contains` to `return isExist(header.next, x);` and runs the check again, jMoped reports that `error()` is executed. Calling "`jmoped -t -`$n$ `LinkedList`" we obtain that no non-terminating execution was detected. Witness paths can also be printed in order to help finding bugs.

The main advantage of jMoped's translator is that SPDSs, its target language, closely matches Java bytecodes. In particular, `invoke` and `return` instructions are directly translated into push/pop SPDS rules. No inlining of bytecodes (which may yield an exponential blowup in size) and no artificial bound on the maximal depth of the stack of method calls are required. The only restrictions are on the data side: Moped requires a bound on the range of variables, and on the maximal number of objects that can be generated.

* This work is supported by the EPSRC Grant GR/93346 "An Automata-theoretic Approach to Software Model Checking" and by the DFG project "Algorithms for Software Model Checking"

## 2 The Translation

We recall that a Java program is compiled into a class file containing bytecodes, the machine language of the Java Virtual Machine (JVM). Bytecodes of the form `invokestatic <name>` or `invokevirtual <name>` invoke the method `<name>`.

A pushdown system consists of a set of *control states*, a *stack alphabet*, and a number of *rules*, which correspond to the well-known transition rules of pushdown automata. An SPDS is a pushdown system together with two sets of *global* and *local* variables over a finite domain. Loosely speaking, there is one single copy of a global variable, but each stack symbol owns a copy of each local variable. The rules of an SPDS are best explained by means of an example. The rule

```
q1 <f1> --> q2 <f2 f3> (x > 3 & y' = x'' + 1)
```

where $x$ and $y$ are local variables, is read as follows: If the current control state is `q1`, the topmost stack symbol is `f1`, and the value of the copy of `x` owned by `f1` is greater than 3, then move to control state `q2`, replace `f1` by `f2 f3` on the stack, and set the copy of `y` owned by `f2` to 1 plus the value of the copy of `x` owned by `f3`. The stack is useful whenever the front end is a procedural language. The local variables owned by a stack symbol correspond to the local variables of a procedure or method. A procedure call and a return are modeled by a push and a pop, respectively.

Our translator first fetches the bytecodes of the methods in the class, and of the methods from other classes called by them. Then, each bytecode instruction is directly mapped into one or more SPDS rules. The JVM uses two stacks: a local stack for each method, whose maximal size is determined at compile time, and the stack of method calls. The local stacks are modeled with *stack variables* called (`sv0,...,sv`*k-1*), where $k$ is the maximum stack depth (usually a low number) obtained from the Java compiler. `sv0` represents the top of the stack. A push of number 1 to the local stack is modeled by a rule of the form

```
q <f1> --> q <f2> ((sv0'=1) & (sv1'=sv0) & (sv2'=sv1) & ...)
```

Figure 1 shows fragments of a Java program and some corresponding bytecodes. Method `contains` checks if the list contains a given value. It calls the recursive method `isExist`. The methods `insert` and `error` (both omitted) add a node to the list and handle errors, respectively.

The translator produces a set of SPDS variables and a set of rules. Roughly speaking, the SPDS variables are the stack variables mentioned above, local SPDS variables matching the variables of a method, plus SPDS variables used to store the values of object fields. The translator assigns an *id* to each object reference created by a `new` bytecode. For every object field, an array of global SPDS variables is created with *id*s as array indices. In our example, the bytecode of the main method creates four references (one for the list `l` and three for nodes). The translator assigns *id*s 1 to 4, and creates three arrays: `header[1,1]`, `value[2,4]`, `next[2,4]`. The numbers in brackets indicate the array bounds.

We can now explain the meaning of the `-`*n* option when calling jMoped: it specifies the number of bits assigned by Moped to each variable, and so its range.

Figure 2 shows some SPDS rules produced by the translator. Notice that the JVM assigns to each method a set of variables indexed by 0,1,2,.... The translator creates the corresponding SPDS variables `var0`, `var1`, etc. The first line of the figure corresponds to bytecode 0: and pushes the value of `var1` onto the local stack. The translation of 1: and 16: should be self-explanatory. The translation of 25: uses a variable `ret` to store the result of the method call. The translation of 46: in method `main` uses a push to model the method invocation and shows how the return value is evaluated.

```
public class ListNode {
  int value;
  ListNode next;
  public ListNode(int x)
   { value = x; next = null; }
}

public class LinkedList {
  private ListNode header;
  public LinkedList()
   { header = null; }
  ...
  public boolean contains(int x)
   { return isExist(header, x); }
  boolean isExist(ListNode n, int x) {
   if (n == null) return false;
   if (n.value == x) return true;
   else return isExist(n.next, x);
  }
  ...
  public static void main(String[] args) {
   LinkedList l = new LinkedList();
   l.insert(new ListNode(1));
   l.insert(new ListNode(2));
   l.insert(new ListNode(3));
   if (!l.contains(1))
    error();
  }
}
```

```
isExist(ListNode, int):
   0: aload_1
   1: ifnonnull +5
   4: iconst_0
   5: ireturn
   6: aload_1
   7: getfield <value>
  10: iload_2
  11: if_cmpne +5
  14: iconst_1
  15: ireturn
  16: aload_0
  17: aload_1
  18: getfield <next>
  21: iload_2
  22: invokevirtual <isExist>
  25: ireturn
main(String[]):
   0: new <LinkedList>
   3: dup
   4: invokespecial <init>
   7: astore_1
      ...
  44: aload_1
  45: iconst_1
  46: invokevirtual <contains>
  49: ifne +6
  52: invokestatic <error>
  55: return
```

**Fig. 1.** Java code (left) and some of its bytecodes (right).

## 3 jMoped and Alloy

We compare our approach to the one of Vaziri and Jackson [3] using the Alloy system. There, Java code is translated into a SAT formula. This requires bounds not only on the range of variables and the number of generated objects, but also on the maximum depth of the call stack and on the number of times a loop can be executed. Moreover, method calls are dealt with by inlining. Our approach removes the last two bounds while staying within a decidable problem [4].

```
Some transition rules of isExist(ListNode, int):
  q <f0> --> q <f1>       ((sv0'=var1) & (sv1'=sv0) & (sv2'=sv1) & ...)
  q <f1> --> q <f6>       ((sv0!=0) & (sv0'=sv1) & (sv1'=sv2) & ...)
  q <f16> --> q <f17>     ((sv0'=var0) & (sv1'=sv0) & (sv2'=sv1) & ...)
  q <f25> --> q <>        ((ret'=sv0) & ...)
Some transition rules of main(String[]):
  q <m46> --> q <c0 m49a> ((var1'=sv0) & (var0'=sv1) & (sv0''=sv2) & ...)
  q <m49a> --> q <m49>    ((sv0'=ret) & (sv1'=sv0) & (sv2'=sv1) & ...)
```

**Fig. 2.** Part of the translation of the code of Figure 1.

The bounds on the range of variables mean that both tools check the presence or absence of errors *within these bounds*. So, in fact, they are carrying out a sort of extended symbolic testing, in which many different test cases (often billions) are checked in one single symbolic computation, and in which non-termination can be explicitly detected.

We have considered the faulty insertion algorithm in red-black trees used in [3], and invariant properties. In [3] the property is written in Alloy, while we added an `error()` method that is executed when the invariant is violated. Vaziri and Jackson report being able to automatically find the bug when the number of nodes and the number of iterations of a loop are both limited to five in 18 seconds. In our case it took 10 seconds in a standard PC. While the two numbers are not directly comparable, because different machines were used, they indicate that our approach, while providing a more direct match to the structure of imperative languages, does not necessarily lose efficiency.

## 4 Conclusion and Future Work

We have described jMoped, a tool that allows to check invariant properties and termination of Java code by translating Java bytecodes into pushdown systems and then applying the Moped tool. The tool covers a large fragment of Java programs. Our ultimate goal (from which we are still far away) is to develop a checking tool for Java programs with a very simple interface that could be used routinely, even by people without a background on verification.

## References


1. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, TU Munich (2002)
2. Suwimonteerabuth, D.: Verifying Java bytecode with the Moped model checker. Master's thesis, University of Stuttgart (2004)
3. Vaziri, M., Jackson, D.: Checking properties of heap-manipulating procedures with a contraint solver. In: TACAS'03. LNCS 2619, Springer (2003) 505–520
4. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Proc. CAV'01. LNCS 2102, Springer (2001) 324–336