Bytecode Model Checking: An Experimental Analysis*

David Basin[1], Stefan Friedrich[1],
Marek Gawkowski[1], and Joachim Posegga[2]

[1] Institut für Informatik,
Universität Freiburg, Germany
[2] SAP AG, Corporate Research,
Karlsruhe, Germany

Abstract. Java bytecode verification is traditionally performed by a polynomial time dataflow algorithm. We investigate an alternative based on reducing bytecode verification to model checking. Despite an exponential worst case time complexity, model checking type-correct bytecode is polynomial in practice when carried out using an explicit state, on-the-fly model checker like Spin. We investigate this theoretically and experimentally and explain the practical advantages of this alternative.

1 Introduction

Java is a popular programming language that is well suited for building distributed applications where users can download code that they can locally execute. To combat the security risks associated with mobile code, Sun has developed a security model for Java and a central role is played by bytecode verification [7, 18], which ensures that no malicious code is executed by a Java Virtual Machine (JVM). Bytecode verification takes place when loading a Java class file and the process verifies that the loaded bytecode program has certain properties that the interpreter's security builds upon. By checking these properties statically before execution, the JVM can safely omit the corresponding runtime checks.

In this paper we show how model checking can be used to check type safety properties (which is the essential, and non-trivial, part of bytecode verification) of Java class files. We have built a system for this task, shown in Figure 1. The system takes as input a Java class file as well as a specification of an abstraction of the Java virtual machine. From these, the system produces an intermediate representation consisting of a transition system (for the abstracted machine) and a specification of safety properties formalizing conditions sufficient for bytecode type safety. Afterwards, these descriptions are translated into the input language of public domain model checkers, currently the Spin [4] and SMV [5] systems.

* The research presented in this paper was partially funded by T-Systems Nova GmbH in the ByCoMoChe project.

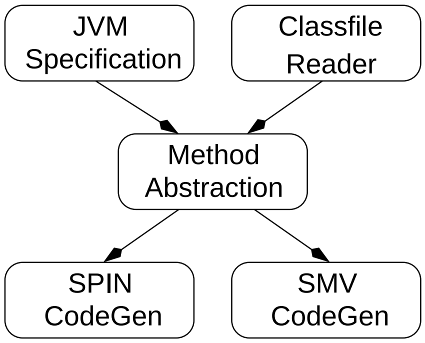


Fig. 1. Overall System Architecture

Motivation and Contributions

There are three reasons why we focused on this problem. First, we were interested in using formal methods to improve the security of the entire verification and execution process. In formalizing the Java Virtual Machine and its abstraction, we show how to explicitly model bytecode verification in a precise language that makes explicit the exact nature of the bytecode verification problem. This goes far beyond the Java documentation [7], which gives only a semi-formal description of bytecode verification and leaves numerous aspects of both the bytecode verifier and the Java Virtual Machine either ambiguous or underspecified. Moreover, by basing a bytecode verifier directly on a general model checker, our approach reduces the chance of errors arising during an implementation of specialized dataflow algorithms. It is noteworthy in this regard that, as Stärk and Schmid point out [16], there are legal Java programs whose compiled bytecode is type correct but are not accepted by Sun's bytecode verifier. The classes of programs that they define (based on calling subroutines in different contexts) are unproblematic for our approach.

Second, we were interested in determining whether this approach is practical and scales to realistic examples. The answer is not obvious. In contrast to conventional bytecode verification, which is based on dataflow analysis and has polynomial time complexity, using a model checker for this task has a time complexity that is exponential in the worst case. We show that for type correct bytecode, model checking can yield results comparable to dataflow procedures. The reason for this is that despite the exponential number of states, for correct code, only polynomially many are reachable (see Section 3 for the exact analysis). Our experiments validate that the use of Spin, which constructs the state space incrementally, on-the-fly, produces results in agreement with these bounds. This is in contrast to symbolic BDD-based model checking approaches like SMV that must represent the entire state space and therefore turn out to be impractical for this kind of problem. For incorrect bytecode, both explicit state and symbolic methods may fail to terminate (or exhaust memory). This is not an issue in practice; when too many resources are used, one may either time out (giving a conservative answer) or use an alternative approach (e.g., property simplification, as described in Section 3.3) to detect an error.

Our result suggests the usefulness of bytecode verification by model checking, especially in domains where time and space requirements are less important than correctness and possible extensibility. One such application is Java for smart cards (JavaCard). The JavaCard is a highly secure smart card platform with a simplified JVM. Due to memory limitations, bytecode verification must be performed off-card (where correct code can then be digitally signed by the verifier), instead of by the run-time environment. Our original motivation for this work was to investigate whether model checking could be used as an alternative in this domain, a question we can now answer positively.

There is a final, independent reason why we think this is an interesting problem: there is an unlimited supply of scalable, real life examples. The Java distribution, for example, comes with thousands of class files that we could use for testing our system. Indeed, for this reason, we would like to suggest bytecode verification as a problem domain to be generally used to test and compare different model checkers. Our system is freely available for such benchmarking purposes.

Related Work

The widespread use of Java and the lack of formal treatment originally given in [7] have motivated considerable research. A number of different approaches have been proposed for type checking bytecode and [6] contains an excellent overview of the area. Most of this work is theoretically oriented and is concerned with formalizing the JVM [2] and defining related type systems [3, 12, 13, 17]. There has also been considerable work on formally proving the soundness of various approaches or verifying sufficient conditions for bytecode verifiers to be correct [2, 8, 10, 11].

In the recent years there has been a convergence of ideas in static analysis and model checking: different kinds of program analysis can be performed by fixed-point computations and these computations can either be carried out by specialized algorithms or by general purpose model checkers [14, 15]. Whereas static analysis techniques have a longstanding history, the application of model checking to static analysis problems is more recent. The idea of using model checking for bytecode verification was originally suggested by Posegga and Vogt [9]. They carried out a few small examples by hand to suggest how, in principle, this approach could work for a subset of the JVM. Our work represents the first large scale effort to apply model checking to this problem and to study its practical significance.

```
class Test {

int fac (int i) {
    if ( i == 0 )
      return 1;
    else
      return i*fac(i-1);
  }
}
```

```
  Method int fac(int)
>> max_stack=4, max_locals=2 <<
    0 iload_1
    1 ifne 4
    2 iconst_1
    3 ireturn
    4 iload_1
    5 aload_0
    6 iload_1
    7 iconst_1
    8 isub
    9 invokevirtual <Test.fac(int):int>
   10 imul
   11 ireturn
```

Fig. 2. Java code and Bytecode of method fac

Organization

The remainder of this paper is organized as follows. In Section 2 we explain the reduction of bytecode verification to model checking. Afterwards, in Section 3 we present experimental results and an analysis. We draw conclusions in Section 4.

2 Abstracting Classfiles to Model Checking Problems

2.1 Background

In this section we briefly explain the bytecode verification problem and describe the main elements of our approach.

Bytecode and the JVM. Java programs are compiled to bytecode instructions that are interpreted by the JVM (see Figure 2 for a small example). The result of compilation, a classfile, contains a symbol table (called the constant pool) describing the fields of the class and a list of the methods of the class. The JVM supports object orientation and there are specific bytecode instructions for generating and accessing the objects of a class. The overall architecture is that of a stack machine: the JVM possesses an operand stack, which is used for the evaluation of expressions. For instance, an `imul` instruction multiplies the topmost two elements of the operand stack, discards those elements from the stack, and pushes the result of the multiplication back on the operand stack. In addition to the stack, the JVM also uses an array of registers to store local variables.

Most JVM instructions are typed. For instance, the `getfield` $C.f.\tau$ instruction, which accesses the field f of type τ in class C, requires that the operand stack contains a reference to an object of class C (and not, for instance, an integer, which would correspond to an attempt to forge a reference). The operand stack and the registers (local variables) however are not typed.

Bytecode verification. To guarantee the secure operation of the JVM, one must show that each method is well-typed, i.e., that one can assign a state type to each point in the program. The state type specifies what kind of values the operand stack and the local variables may contain at the given program point. For example, the state type $(0, \mathsf{Empty}, loc[0 \mapsto \mathsf{REF}(Test), 1 \mapsto \mathsf{INT}])$ associates to the program point 0 those states with an empty operand stack whose first local variable contains an object of class *Test* and whose second local variable contains an integer. Given this notion of a well-typing, one can show that the execution of a well-typed method will never lead to a bad state of the JVM, that is, a state where the instructions operate on inappropriate data. This allows the JVM to execute more efficiently by eliminating runtime type checks.

Conventional bytecode verification. Conventional bytecode verification works by abstracting a method to a state transition system and then computing a type of the method by dataflow analysis. The bytecode verifier checks, on-the-fly, that the computed type is a well-typing, i.e. satisfies the conditions for the correct execution of the instructions.

Type construction is carried out by computing the supremum of the state types of all possible execution paths. This requires the unique existence of such a supremum, which is however, in the presence of multiple inheritance (interfaces), not the case. There are several solutions to address this problem, including considering sets of types instead of single types or checking correct implementation of interfaces at runtime (as done by SUN's bytecode verifier).

Conventional bytecode verification is further complicated by the existence of subroutines, which are used to compile the `finally` part of a Java `try-catch-finally` construct. The complication is due to the fact that one can call (`jsr`) and return from (`ret`) subroutines from different program points where the calling contexts (the stack and the register values not used by the subroutine) of different execution paths can be incompatible. This results in non-trivial complications; solutions include structural restrictions on bytecode (Sun's approach) and polyvariant dataflow analysis [6]. It is an open question which solution is best. The polyvariant approach seems more elegant but has a time complexity that is exponential in the depth of subroutine nesting.

Model checking approach. In our approach we also abstract a method to a state transition system. However, instead of performing a dataflow analysis, we formalize the correctness properties as predicates of the states of the abstract transition system and use an off-the-shelf model checker like Spin or SMV to check that these properties are satisfied. The model checker then either reports the correctness of the method, or it provides a counter example in form of an execution trace that leads to a failure.

This approach is simpler than conventional bytecode verification, as the requirements for the correctness can be clearly and comprehensibly formalized in an easy to understand language, which helps to avoid errors in the formalization. Moreover, with our approach, multiple inheritance is not problematic since the formalization of the correctness properties requires only that the types are ordered.

Note further that the problems alluded to above concerning subroutines do not arise in our approach since model checkers consider all possible runs of the system; they are, in the words of [6, p281], "the ultimate polyvariant analysis". However, the cost is a time complexity exponential in the depth of subroutine nesting. We investigate this problem and its implications in Section 3.2.

2.2 Abstraction/Transition System

We abstract a method M to a finite state transition system (Q, q_0, Δ). The set $Q \subseteq \mathbb{N} \times (\mathcal{T}\ \mathsf{stack}) \times (\mathcal{T}\ \mathsf{array})$ of states contains triples that consist of the program counter, the operand stack, and the array of local variables of the method M. The set $\mathcal{T}$ of types contains the primitive types and the reference types of the JVM (NULL represents the polymorphic type of the null reference) and the program addresses that can be targets of `ret` instructions. We add an element UNDEF to represent uninitialized values.

$$
\begin{aligned}
prim &= \{\mathsf{INT}, \mathsf{FLOAT}, \mathsf{LDOUBLE}, \mathsf{HDOUBLE}, \mathsf{LLONG}, \mathsf{HLONG}\}\\
ref &= \{\mathsf{REF}(cn) \mid cn \in classnames\} \cup \{\mathsf{NULL}\}\\
adr &= \{\mathsf{ADR}(i) \mid i \in \mathbb{N}\}\\
\mathcal{T} &= \{\mathsf{UNDEF}\} \cup prim \cup ref \cup adr
\end{aligned}
$$

Since only a finite subset of $\mathcal{T}$ occurs in a particular transition system, we can compute this subset by inspecting the signature and the method body. The signature of a method, which has the form

$$S = (M, result\_type, [arg\_type_1, \ldots, arg\_type_n])$$

specifies the method's name, its argument types and its result type. The method body is given as a list ins of bytecode instructions. The various instructions in the method body also introduce new types, e.g.,

$$
\begin{aligned}
&\mathsf{types\_of\_ins}(\texttt{imul}) &&= \{\mathsf{INT}\}\\
&\mathsf{types\_of\_ins}(\texttt{new}\ C) &&= \{C\}\\
&\mathsf{types\_of\_ins}(\texttt{getfield}\ C.f.\tau) &&= \{C, \tau\}\\
&\qquad\qquad\vdots && .
\end{aligned}
$$

Thus the set T of types occurring in a method that belongs to class C is

$$
\begin{aligned}
T = &\{\mathsf{REF}(C), result\_type, arg\_type_1, \ldots, arg\_type_n\} \cup\\
&\bigcup_{p \in \{0, \ldots, |ins|-1\}} \mathsf{types\_of\_ins}(ins_p).
\end{aligned}
$$

We compute the initial state q_0 of the transition system for a method M with signature S that belongs to a class C as follows: Execution starts at program counter 0 with an empty operand stack, the `this` reference, and the actual parameters, which are passed through the first $n+1$ local variables. The remaining local variables $loc[n+1], \ldots, loc[maxloc]$ are initially undefined ($maxloc$ is specified in the classfile), i.e.,

$$q_0 = (0, \mathsf{Empty}, loc), \text{ where } \begin{cases} loc[0] = \mathsf{REF}(C) \\ loc[1] = arg\_type_1, \ldots, loc[n] = arg\_type_n \\ loc[n+1] = \mathsf{UNDEF}, \ldots, loc[maxloc] = \mathsf{UNDEF}. \end{cases}$$

The transition relation of the abstract method is defined by the instructions of the method. We give here a few representative examples, which show how the program counter pc, the operand stack $opst$, and the local variables loc are modified:

istore n	$(pc, opst, loc) \mapsto \{(pc+1, opst, loc[n \mapsto \mathsf{top}(opst)])\}$
iload n	$(pc, opst, loc) \mapsto \{(pc+1, loc[n].opst, loc)\}$
imul	$(pc, opst, loc) \mapsto \{(pc+1, \mathsf{INT}.\mathsf{pop}(\mathsf{pop}(opst)), loc)\}$
new C	$(pc, opst, loc) \mapsto \{(pc+1, \mathsf{REF}(C).opst, loc)\}$
getfield $C.f.\tau$	$(pc, opst, loc) \mapsto \{(pc+1, \tau.\mathsf{pop}(opst), loc)\}$
ifeq *offset*	$(pc, opst, loc) \mapsto \{(pc+1, \mathsf{pop}(opst), loc), (pc+offset, \mathsf{pop}(opst), loc)\}$
jsr *offset*	$(pc, opst, loc) \mapsto \{(pc+offset, \mathsf{ADR}(pc+1).opst, loc)\}$
ret n	$(pc, opst, loc) \mapsto \{(\mathsf{retaddr}(loc[n]), opst, loc)\}$

2.3 Type Safety Properties

We formalize correctness properties as predicates on the states of the abstract transition system. These properties must hold globally for all possible runs of the system and this motivates our use of a temporal specification formalism. Two different kinds of properties are required.

First, the operand stack must not overflow. The operand stack is only used to evaluate expressions and hence the maximal stack-height $maxstack$ can be computed in advance. This value is given as a method attribute in the classfile of the method. We can formulate the corresponding condition as $\mathsf{size}(opst) \leq maxstack$.

Second, each instruction must always operate on data of the appropriate type. Note that due to object orientation, for some instructions different types of data are acceptable as determined by the subtyping relation $\sqsubseteq_\Gamma$, which depends on the program Γ. Figure 3 shows the subtyping relation $\sqsubseteq_\Gamma$ for a program Γ consisting of three classes C, D, and E. In the specification of the JVM, the well-typing conditions can be formalized in a straightforward, declarative way.

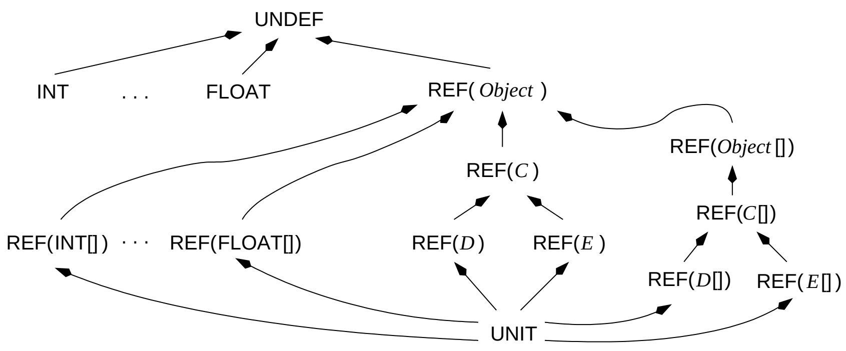


Fig. 3. Subtyping relation $\sqsubseteq_\Gamma$

We give here a few representative examples:

$$\begin{array}{ll}
\texttt{aload}\ n & \mapsto loc[n] \in \mathit{ref} \cap T \\
\texttt{iload}\ n & \mapsto loc[n] = \mathsf{INT} \\
\texttt{imul} & \mapsto (\mathsf{top}(opst) = \mathsf{INT})\ \wedge\ (\mathsf{top}(\mathsf{pop}(opst)) = \mathsf{INT}) \\
\texttt{getfield}\ C.f.\tau & \mapsto \mathsf{top}(opst) \sqsubseteq_\Gamma \mathsf{REF}(C) \\
\texttt{putfield}\ C.f.\tau & \mapsto \mathsf{top}(\mathsf{pop}(opst)) \sqsubseteq_\Gamma \mathsf{REF}(C)\ \wedge\ \mathsf{top}(opst) \sqsubseteq_\Gamma \tau
\end{array}$$

For a given program Γ, the relation $\sqsubseteq_\Gamma$ is finite and thus the conditions can be unfolded and automatically checked. The condition for the `getfield` $C.f.\tau$ instruction, for instance, would be unfolded to

$$\begin{array}{rcl}
\mathsf{local_cond}(\texttt{getfield}\,C.f.\tau) & \equiv & \mathsf{top}(opst) = \mathsf{REF}(C)\ \vee\ \mathsf{top}(opst) = \mathsf{REF}(D)\ \vee \\
 & & \mathsf{top}(opst) = \mathsf{REF}(E)\ \vee\ \mathsf{top}(opst) = \mathsf{UNIT}.
\end{array}$$

The overall correctness property for a method is the conjunction of the global property for the stack height and the local property for each program point.

$$(\mathsf{size}(opst) \leq maxstack) \wedge \bigwedge_{p \in \{0,\ldots,|ins|-1\}} (pc = p \Rightarrow \mathsf{local\_cond}(ins_p))$$

2.4 Backends for Spin and SMV

We have implemented two different backends: one for Spin and one for SMV. The basic idea in both is the same. From our intermediate representation we produce a transition system in the input language of the model checker and a property specification. The property specification states globally invariant correctness properties, i.e., properties that must hold at every program point. In LTL this corresponds to checking $\Box\varphi$ for a state property φ and in CTL this corresponds to checking $\mathsf{AG}(\varphi)$.

We briefly describe here the Spin backend (SMV is similar in most respects). The formalization of the transition system in Spin's input language Promela

```
init { atomic { loc[0]=15; loc[1]=13; opst_ptr=0; pc=0 };
       run assertions ();  run transitions () }

proctype transitions( ) {
  do
  :: pc==0 -> atomic { opst[opst_ptr]=loc[1]; opst_ptr=opst_ptr+1; pc=1 };
  :: pc==1 -> if
                :: atomic { opst_ptr=opst_ptr-1; pc=2 };
                :: atomic { opst_ptr=opst_ptr-1; pc=4 }
               fi;
  :: pc==2  -> atomic { opst[opst_ptr]=13;     opst_ptr=opst_ptr+1; pc=3 };
  :: pc==3  -> atomic { break };
  :: pc==4  -> atomic { opst[opst_ptr]=loc[1]; opst_ptr=opst_ptr+1; pc=5 };
  :: pc==5  -> atomic { opst[opst_ptr]=loc[0]; opst_ptr=opst_ptr+1; pc=6 };
  :: pc==6  -> atomic { opst[opst_ptr]=loc[1]; opst_ptr=opst_ptr+1; pc=7 };
  :: pc==7  -> atomic { opst[opst_ptr]=13;     opst_ptr=opst_ptr+1; pc=8 };
  :: pc==8  -> atomic { opst[opst_ptr-2]=13;   opst_ptr=opst_ptr-1; pc=9 };
  :: pc==9  -> atomic { opst[opst_ptr-2]=13;   opst_ptr=opst_ptr-1; pc=10 };
  :: pc==10 -> atomic { opst[opst_ptr-2]=13;   opst_ptr=opst_ptr-1; pc=11 };
  :: pc==11 -> atomic { break }
  od
}
```

Fig. 4. Abstract transition system for the *fac* bytecode in Spin

is straightforward. The types of the transition system, i.e. the elements of the set T, are represented as integers. The stack and the array of local variables are modeled as arrays of integers. As an example, Figure 4 shows the abstract transition system for the *fac* bytecode presented in Section 1. The data required to model this method are the address labels $\{\mathsf{ADR}(0), \ldots, \mathsf{ADR}(11)\}$ and the types $\{\mathsf{UNDEF}, \mathsf{INT}, \mathsf{UNIT}, \mathsf{REF}(\mathit{Test})\}$. They are represented (in this order) by the numbers 0 through 15. Initially the operand stack is empty, the local variable $loc[0]$ contains the *this* reference $\mathsf{REF}(\mathit{Test})$ and the local variable $loc[1]$ contains the INT argument of the method. Each transition modeling an instruction is carried out as an atomic step. A branching instruction, which produces two possible successor states, is modeled using a nondeterministic if-statement.

Note that in Spin the invariant φ can be expressed in different ways, e.g., as an observer process or using a never-claim. We have chosen the former: The observer process runs in parallel to the abstract state transition system and it contains an assertion statement that states the correctness property that must hold at each state. This approach is simple and has the practical advantage that temporal formulas need not be translated separately to automata. Figure 5 shows the correctness properties for our sample bytecode. For example, for the `invokevirtual<Test.fac(int):int>` instruction at $pc = 9$, we require that the topmost element of the operand stack is an INT (i.e.

```
proctype assertions() {
  assert (opst_ptr<=4 &&
          (pc!=0  || loc[1]==13) &&
          (pc!=1  || opst[opst_ptr-1]==13) &&
          (pc!=3  || opst[opst_ptr-1]==13) &&
          (pc!=4  || loc[1]==13) &&
          (pc!=5  || loc[0]==14 || loc[0]==15) &&
          (pc!=6  || loc[1]==13) &&
          (pc!=8  || opst[opst_ptr-2]==13 && opst[opst_ptr-1]==13) &&
          (pc!=9  || opst[opst_ptr-1]==13 && opst[opst_ptr-2]==15) &&
          (pc!=10 || opst[opst_ptr-2]==13 && opst[opst_ptr-1]==13) &&
          (pc!=11 || opst[opst_ptr-1]==13) )
}
```

Fig. 5. Correctness properties for *fac*

`opst[opst_ptr-1]==13`) and the next element is a reference to an instance of the class *Test* (i.e. `opst[opst_ptr-2]==15`).

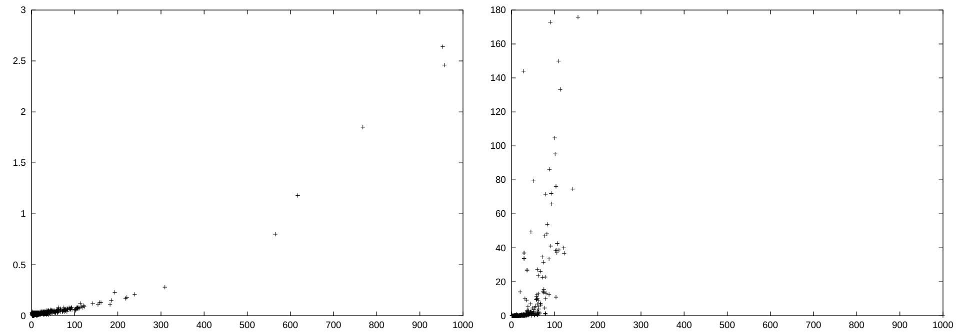
We will not provide here a formal proof of the correctness of the translation. However, the basis is given in the work of [11] where they specify sufficient conditions for Java bytecode verifiers to be correct. The main idea is that a method is well-typed when it can be assigned a type (composed of state types, described in Section 2.1) and the existence of such a well-typing guarantees that method execution proceeds without type errors (stack overflow or improper arguments to instructions). It is possible to show that when a method is successfully model checked in our setting then such a well-typing exists.

3 Experimental Results and Analysis

We have carried out two different kinds of experiments to investigate the applicability and scalability of using model checking for bytecode verification. First, to test the practical applicability of this approach, we model checked all the methods associated with a large Java library, namely all methods of the `java.lang` package. Second, to better understand how the complexity of bytecode checking depends on parameters such as method length, nesting of subroutines, or number of variables and the types involved, we carried out systematic "stress tests" where we varied each parameter individually, while leaving the others fixed.

3.1 Practical applicability

To test the applicability of our approach to the verification of real bytecode, we tested it on all the methods of the `java.lang` package. This package contains 109 classes with 841 methods. These methods are representative for Java bytecode as they contain all the instructions of the JVM. Moreover, they vary considerably in their complexity in terms of the different parameters mentioned above. Also representative is their size. Most Java methods in practice are modestly sized. In this package, only 32 methods contain more than 100 instructions.



(a) SPIN (b) SMV

Fig. 6. Verification times depending on the number of instructions in a method

As Figure 6 indicates, SPIN checks almost all of these methods in negligible time.[1] Only the largest five, which each contain more than 600 instructions, require more than half a second. This is sufficient for the most applications and in particular is more than adequate for off-line verification (e.g., smart cards) and for applications with on-line verification, such as web applets, where methods are generally quite small. The picture for SMV is completely different; even small methods with less than 100 instructions can require more than two minutes to verify and, for 29 methods, SMV ran out of memory. This suggests that, for this problem domain, explicit state on-the-fly methods are superior to symbolic methods. Our systematic tests shed light on some of the reasons for this.

3.2 Systematic tests

We carried out four different "stress tests" to isolate and investigate the influence of various parameters on the complexity of the model checking problem. In particular, we investigated the effects of individually varying the following:

1. The length of the method, i.e. the number of instructions,
2. the number of local variables,
3. the depth of the class hierarchy, i.e. the number of types that are used to model a method, and

[1] In all experiments, times are measured in seconds. All timings are performed on an 800-Megahertz Pentium III PC with 256 MB memory.

```
static int z;

public static int m_n (int x) {
  z = z + x;
     ⋮            (n times)
  z = z + x;
return z; }
```

Fig. 7. Schema of methods to test dependency of complexity on code length

4. the depth of the nesting of subroutines.

The methods checked were automatically produced by generating and compiling appropriate Java programs as explained below.

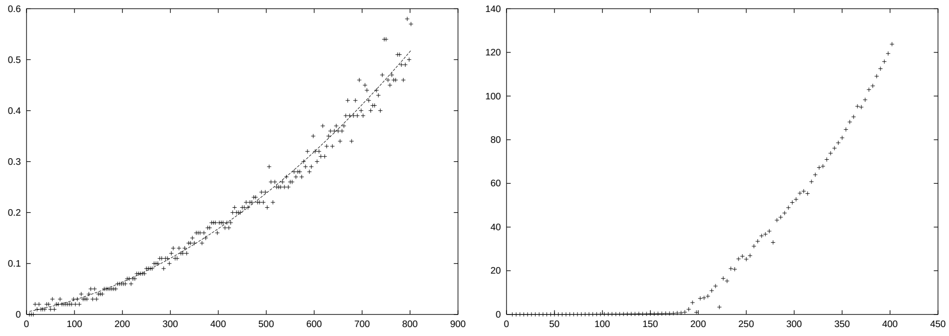
Method length. We investigated the influence of a method's length by generating methods that consist of a single expression repeated n times, for $n \in \{1, \ldots, 100\}$. Figure 7 shows the form of these methods. The corresponding bytecode uses only one local variable `x` and one class variable `z`. The method is declared in a static class that is a direct subclass of `Object`; thus the class hierarchy has depth one. The repeated line of code, `z = z + x`, is translated to a sequence of four bytecode instructions.

As Figure 8 shows, SPIN runtimes scale roughly quadratically with the size of the method and the associated constant factors are small enough to allow practical large scale verification. In contrast, SMV has acceptable verification times until a threshold of around 200 instructions, and then scales quite poorly. It may be possible to delay this threshold by tuning different system specific parameters (e.g., the size of hash tables). However, the symbolic representation of the entire state space in SMV can lead to memory problems, even with good variable orderings, and results in runtimes orders of magnitude larger than SPIN's. Perhaps surprisingly, these problems appear in even such a simple test.

Number of variables. In this test, we varied the number of local variables, which also constitute the arguments of the method. To insure that all variables are used, the method body simply cyclically permutes the contents of the variables. This is done in such a way that the number of assignments, and thus the number of instructions, is the same in each method.

Figure 10 displays the results. For SPIN, the number of variables has little effect on the explicit state-space exploration: the time consumed grows very slowly as a function of the number of variables.[2] The slight variations are caused by the time required to initialize the transition system (that is, SPIN's `init` routine,

[2] Due to limited timing accuracy, this slow growth manifests itself in Figure 10 in a subtle way. Namely, almost all times lie between 0.06 and 0.08 seconds, but the concentration at 0.07 and 0.08 is higher as more variables are added.



(a) SPIN (b) SMV

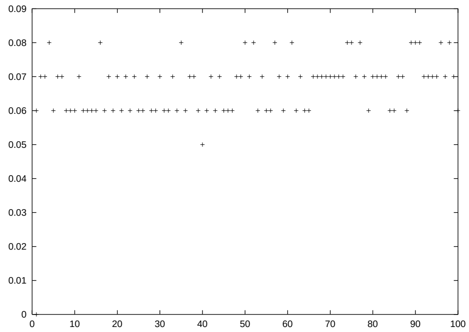
Fig. 8. Verification time depending on the number of instructions

```
void m_1(int v0)    void m_2(int v0, int v1)   void m_3(int v0, int v1, int v2)
{ v0 = v0;          { v0 = v1;                 { v0 = v1;
  v0 = v0;            v1 = v0;                   v1 = v2;
  v0 = v0; }          v0 = v1; }                 v2 = v0; }
```

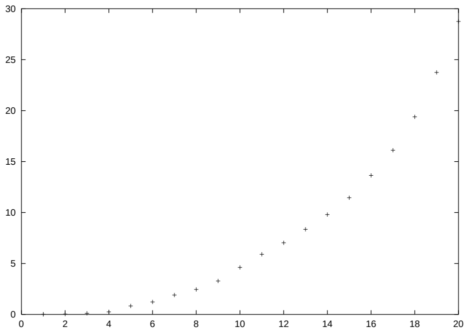
Fig. 9. Schema of methods to test dependency of complexity on the number of local variables

which assigns to each variable its initial value), which is linear in the number of variables. Apart from this, the time required is constant as only a constant set of states (one for each program point, cf. Section 3.3) is reachable. In contrast, SMV's time complexity grows rapidly in the number of variables and quickly becomes impractical with more than 20 variables. This is not too surprising since SMV must represent and manipulate the complete transition system. Symbolic representation using BDDs provides some help here: As suggested by Figure 11 (logarithmic scale for the y-axis), the growth is subexponential since BDDs can compress the representation somewhat. Still, this is, in general, the major bottleneck with symbolic model checking, and it is a substantial problem in this particular domain.

Class hierarchy depth. For this test we generated a linear hierarchy of 100 classes, $C_1, \ldots, C_{100}$, such that C_{i+1} is a direct subclass of C_i. An additional class contains 100 methods, $m_1, \ldots m_{100}$, to be checked. Each of these method takes 100 arguments. The arguments all have the same type in the first method. The second method has arguments of two different types and, in the general case, the ith method's arguments are of i different types. For each method, the method body consists of a single assignment statement. Since these assignments are all identical, the abstract transition system is the same for every method. However,



(a) Spin



(b) SMV

Fig. 10. Verification time versus number of local variables

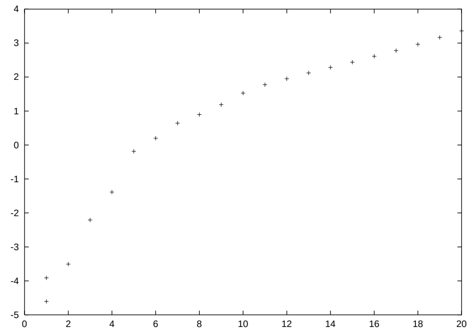


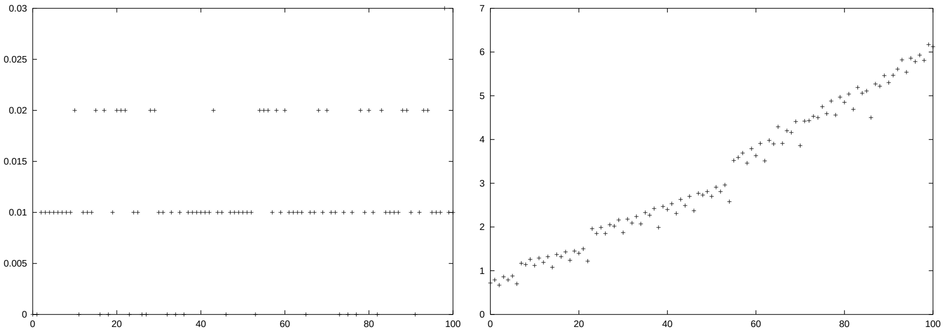
Fig. 11. Verification time versus number of local variables for SMV(logarithmic scale)

the state space grows as more types are present; moreover, the properties to be checked also become more complex. In the case of Spin, the depth of the class hierarchy has no effect on the verification time as the set of reachable states does not change as more types are added (the variation of 0.01 seconds is due to inaccurate timing).

In the case of SMV, the verification time grows linearly with the number of different types. Examining the graph, we can identify four different groups of methods, where the ith group contains methods using 2^i to $2^{i+1}-1$ different class types. The reason for this grouping is that for the ith group SMV requires $i+2$ bits to represent these types and it appears that a small additional amount of time (corresponding to the small gap between the groups) is required to manipulate the larger BDDs. Note that the 100 local variables used do not blow

```
public void m_1   (C_1 c_1, C_1 c_2,  ..., C_1   c_100) {c_1.field = 0;}
public void m_2   (C_1 c_1, C_2 c_2,  ..., C_1   c_100) {c_1.field = 0;}
                                     ⋮
public void m_100 (C_1 c_1, C_2 c_2,  ..., C_100 c_100) {c_1.field = 0;}
```

Fig. 12. Schema of methods for testing dependency of complexity of depth of class hierarchy



(a) Spin (b) SMV

Fig. 13. Verification time versus depth of class hierarchy

up the state space as only c_1 is actually used; that is, in this example the BDDs achieve an exponential compression of the state space.

Depth of subroutine nesting. Our last test considered methods that contain nested subroutines where the depth of the nesting is increased in each method. As explained in Section 2.2, verifying subroutines is one of the more delicate issues in bytecode verification. The fact that verifying subroutines by model checking is much simpler than verifying them conventionally comes at the price of exponential time consumption! Both Spin and SMV do not cope well with checking nested subroutines, as Figure 15 illustrates. Since subroutines are polymorphic in the local variables that are not used in the subroutine, the reachable state space in this case is exponential in the depth of subroutine nesting. BDDs do not have any significant impact on this explosion.

3.3 Analysis

In the following we compare the complexity of conventional bytecode verification with model checking. Let *ins* be the number of program points, T the number

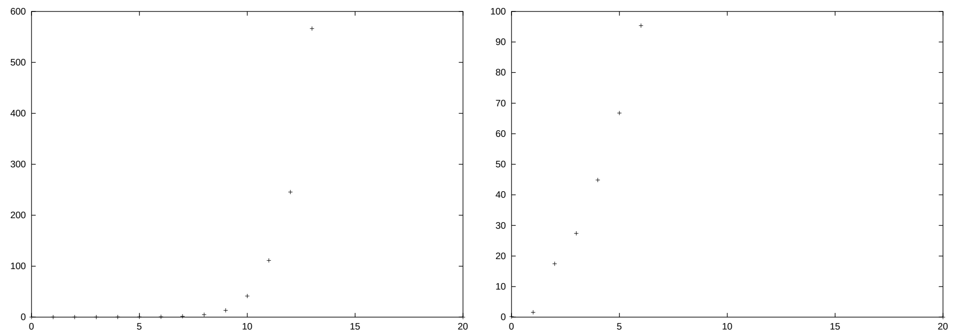
```
public void m_1() {
  int x = 0;

  x++;   // repeated 52 times
  try {throw new Exception();}
  catch (Exception e) {}
  finally {} }
```

```
public void m_2() {
  int x = 0;

  x++;   // repeated 39 times
  try {throw new Exception();}
  catch (Exception e) {}
  finally{
    try{throw new Exception();}
    catch (Exception e) {}
    finally{} } }
```

Fig. 14. Sample method for testing complexity of subroutine nesting



(a) Spin (b) SMV

Fig. 15. Verification time versus depth of the subroutine nesting

of types, *maxloc* the number of local variables, and *maxstack* the maximal stack height. For conventional bytecode verification, the size of the state space is

$$ins \cdot (2^T)^{maxloc+maxstack},$$

as each program point is (due to multiple inheritance) associated with a set of types for the local variables and stack positions. Despite the size of this search space, in conventional bytecode verification, only linearly many states ever need to be explored. In particular, the method type that associates a state type with every program point is computed by iterating an abstract interpretation of the method. The algorithm starts with a method type that associates the bottom type to each program point. In each iteration, the state types belonging to different execution paths are merged by taking their supremum. This yields a new state type for each program point that is larger or equal (under the subtyping order) than that of the previous iteration. This process terminates when a fixed point is reached or an error is found. Structural restrictions placed on bytecode (e.g. no two subroutines can be terminated by the same return instruction) guarantee that the number of iterations required is linear in the number of program points. Since no iteration can decrease the types associated with any program position, a fixed point must be reached in $O(ins \cdot T \cdot (maxloc + maxstack))$ iterations.

In the case of model checking, the state space is also exponential. However, as program points are associated with types, instead of sets of types, the size is

$$ins \cdot T^{maxloc+maxstack} \, .$$

The complexity of model checking itself depends on the algorithm used. Symbolic methods manipulate a representation of the entire state space and, as we have seen, can require exponential resources to do so. However, assuming that the nesting of subroutines has some fixed upper bound (in practice the nesting is almost never greater than two), in correct methods only $O(ins)$ of these states are reachable since there is only one state type possible for each program point.

This tractability result only holds for type-correct bytecode. For incorrect bytecode there can indeed be exponentially many reachable states since any type can be associated with any local variable or stack position at each program point. Preliminary experiments with incorrect bytecode confirm that checking it is considerably more resource intensive than checking correct bytecode. For even small methods consisting of less than 100 instructions, both Spin and SMV are incapable of finding errors; typically Spin fails to terminate and SMV runs out of memory. This is not a problem in practice; when too many resources are used, one may either time out (giving a conservative answer) or use an alternative approach to detect an error.

For detecting errors in incorrect code we have found the following "property simplification" approach useful. Instead of checking the correctness properties for all instructions simultaneously (e.g., the large conjunct in Figure 5), the properties checked are split (divide-and-conquer) into subproperties, which are individually checked in separate model checking runs. In the extreme case, we can individually check the safety of each transition from each possible program point (e.g., perform a model checking run for each conjunct in Figure 5). This trades off space for time, reducing the size of the overall transition system for each run, which is the product of the transition system modeling the method and the transition system representing the properties. This approach has proved adequate for finding type flaws in our tests. Bounded model checking [1] is an interesting possible alternative, as normally the paths to errors are fairly small.

4 Conclusion

Our investigation is the first, realistic, large scale study of bytecode verification by model checking. Moreover, to the best of our knowledge, it is one of the larger case studies in using model checking for static analysis. Our conclusion is that, despite being theoretically intractable in the worst case, model checking is in fact practically viable. The key insight is that for practical applications, validating *correct* code is important; this is feasible since only linearly many states are accessible (provided subroutine nesting is limited, as it is in practice). Our tests confirm that explicit state, on-the-fly model checkers like SPIN can be successfully employed for these kinds of problems; this is in contrast to symbolic model checkers like SMV that must manipulate representations of the entire state space. However, it is open, and an area for further investigation whether alternative encodings of the transition system and the correctness requirements could result in competitive performance.

The system we have implemented can model check full JVM bytecode, i.e., it models all 200 instructions of the JVM. Currently, the only feature missing is code to model object initialization. This has been implemented, but it is not yet completely tested and remains as future work. This issue is rather subtle as explained in [3, 6]. In addition, as future work we would also like to investigate the question of how such a general framework can be used to go beyond model checking type safety properties and validate other kinds of security properties of bytecode.

References

1. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS'99*, volume 1579 of *LNCS*, Amsterdam, the Netherlands, 1999. Springer-Verlag.
2. R. Cohen. The defensive java virtual machine specification. Technical report, Computational Logic Inc., 1997.
3. S. N. Freund and J. C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, Nov. 1999.
4. G. J. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
5. K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1992. CMU-CS-92-131.
6. X. Leroy. Java bytecode verification: An overview. In *Computer Aided Verification, 13th International Conference*, volume 2001 of *LNCS*, pages 265–285, Paris, France, July 2001. Springer-Verlag.
7. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Number 1102 in The Java Series. Addison-Wesley, Reading, MA, USA, Jan. 1997.
8. T. Nipkow. Verified bytecode verifiers. In *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *LNCS*, pages 347–363. Springer-Verlag, 2001.
9. J. Posegga and H. Vogt. Byte code verification for Java smart cards based on model checking. In *Proceedings of the Fifth ESORICS*, volume 1485 of *LNCS*, pages 175–190, Louvain-la-Neuve, Belgium, Sept. 1998. Springer-Verlag.
10. C. Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical Report TUM-I9816, Institut für Informatik, Technische Universiät München, 1998.
11. C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *Tools and Algorithms for the Construction and Analysis of Systems*

(TACAS'99), volume 1579 of *LNCS*, pages 89–103, Amsterdam, the Netherlands, 1999. Springer-Verlag.
12. Z. Qian. A formal specification of Java virtual machine instructions for objects, methods and subroutines. In *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 271–311. Springer-Verlag, 1999.
13. Z. Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Transactions on Programming Languages and Systems*, 22(4):638–672, 2000.
14. D. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Conference record of POPL '98*, pages 38–48, San Diego, 1998. ACM Press.
15. D. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In *Proceedings of Static Analysis Symposium (SAS'98)*, volume 1503 of *LNCS*, pages 351–380, Pisa, Italy, September 1998. Springer-Verlag.
16. R. F. Stärk and J. Schmid. Java bytecode verification is not possible. In *Formal Methods and Tools for Computer Science, Eurocast*. Universidad de Las Palmas de Gran Canaria, 2001. Extended Abstract.
17. R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, Jan. 1999.
18. F. Yellin. Low level security in Java. In *World Wide Web Journal: The Fourth International WWW Conference Proceedings*, pages 369–380, Cambridge, MA, 1995. O'Reilly.