Java Bytecode Verification Using Model Checking*

Joachim Posegga and Harald Vogt

Technologiezentrum
Deutsche Telekom AG
D-64307 Darmstadt, Germany
+49 6151 83 7881
{posegga|vogth}@tzd.telekom.de

Abstract. We provide an abstract interpretation for Java bytecode in order to build finite state models of bytecode programs. The bytecode constraints for assuring safety are formulated in terms of temporal logic formulae. These formulae are checked against the finite program models by a (standard) model checker. By doing so we see a practical way to perform bytecode verification on a formal basis. This could help to achieve higher security and open the door for further extensions to prove additional properties of Java bytecode.

Keywords Java, Java Bytecode Verification, Model Checking

1 Introduction

Java bytecode verification aims at proving that a given bytecode program (e.g., a compiled Java method) conforms to certain security requirements intended to protect the executing platform from malicious code. For reasons of efficiency, the necessary checks are performed once before the program is executed rather than during runtime. Current implementations of this process use techniques from data flow analysis to assure the required properties. Although there exists at least one formalization of the process of bytecode verification ([4]) and one thorough implementation of the bytecode verifier ([10]), there seems to be no implementation which actually is built upon a rigid formal description.

The goal of this work is to show how Java bytecode verification can be implemented as a model checking problem[1]. We are aiming at using a standard model checker tool like SMV [7], thus being able to describe bytecode verification at a very high level in terms of CTL formulae. This gives us a distinct description of the problem and, hopefully, an efficient implementation. The ideas presented in this paper were introduced in [8] in the context of Java smart cards where the

* The opinions expressed in this paper are solely those of the authors and do not necessarily reflect the views of Deutsche Telekom AG.

[1] The idea of implementing a data flow analysis as a model checking problem seems to have been first published in [12].

demand for security is very high and additional checks, even if time–consuming, are desirable.

In this paper, we will not treat all bytecode instructions in every detail but rather concentrate on the `jsr/ret` instructions which are used for implementing subroutines within bytecode programs. This is because they are rarely treated in the literature and they raise special problems for bytecode verification. The paper [11] deals with these instructions by tracking subroutine calls in a stack-like structure. We take up this idea and introduce a structure for recording the active subroutines at each program point.

Bytecode subroutines impose certain difficulties on bytecode verification. First, the variables used by the subroutine must have appropriate types for the subroutine; this limits the use of these variables in the code before the subroutine call. (In our approach, this is easily solved: for each instruction, the type of the used variables is checked the same way, regardless of the path through which the instruction was reached.) Second, the `ret` instruction must be used in a well-structured way such that the variable used by `ret` contains a (valid) return address. Additionally, subroutines may not be called recursively, and several subroutines can be completed by returning from an inner subroutine call. By using a list-like structure to record the active subroutines at each program point, we are able to decide if an execution of `jsr/ret` instructions is possibly violating these constraints.

We start by introducing an abstract interpretation for bytecode instructions and programs in Section 2. This interpretation is finite, allowing us to build a finite state system from a bytecode program which is amenable to model checking. In Section 3, we sketch how to actually build such a system for the SMV model checker from our description. The properties which have to be checked on the finite state system are discussed and we show how they translate to CTL in Section 4.

2 An Abstract Interpretation for Bytecode

2.1 Bytecode Programs

The operational semantics of bytecode instructions is given in terms of transition rules on a state system. We are employing the formalism of Abstract State Machines [5] to describe bytecode programs. Given one rule for every instruction of a bytecode program, we compose a rule for the whole program.

A bytecode program is a sequence of instructions $I_1; I_2; \ldots; I_n$. Given a rule R_I for every bytecode instruction I, the semantics of a bytecode program is given as the (block) rule

$$
\begin{array}{l}
\textit{if } pc = 1 \textit{ then } R_{I_1} \\
\textit{if } pc = 2 \textit{ then } R_{I_2} \\
\ldots \\
\textit{if } pc = n \textit{ then } R_{I_n} \ .
\end{array}
$$

The rules R_{I_i} are instantiations of rule schemes for the respective instructions, where certain placeholders (e.g., denoting the goal of a `jsr` instruction) are substituted by concrete values.

2.2 Semantics of Instructions

We supply a generic ASM rule R_I (a rule scheme) for every bytecode instruction I. A rule R_I is the composition of rules S_I and S'_I, where S_I denotes the rule which gives the "standard" semantics of the bytecode instruction I, whereas S'_I gives an extension to the standard semantics. We will use the latter to explain how the instructions behave on additional state variables, namely the *subroutine call chain* which records the active subroutines. It is required that rules do not *clash*, i.e. the rules S_I and S'_I may not perform contradictory updates. As the rules S'_I make updates on additional state variables only, this is guaranteed.

The standard semantics of bytecode instructions is given by the following rules:

$$
\begin{array}{l}
S_{jsr\ L} \equiv \\
\quad opd := push(opd,\ pc{+}1) \\
\quad pc := L
\end{array}
$$

$$
\begin{array}{l}
S_{ret\ x} \equiv \\
\quad pc := loc(x)
\end{array}
$$

$$
\begin{array}{l}
S_{astore\ x} \equiv \\
\quad loc(x) := top(opd) \\
\quad opd := pop(opd) \\
\quad pc := pc{+}1
\end{array}
$$

The extension to the standard semantics is given by the following rules:

$$
\begin{array}{l}
S'_{jsr\ L} \equiv \\
\quad sr := L \cdot sr
\end{array}
$$

$$
\begin{array}{l}
S'_{ret\ x} \equiv \\
\quad let\ (\_,sr') = split(sr,\ loc(x)) \\
\quad in \\
\qquad sr := sr'
\end{array}
$$

L serves as a placeholder for the goal of the `jsr` instruction. It is replaced by a concrete address (of a program point) when the rule is instantiated to build the ASM for a bytecode program. (The underscore _ in the *let* statement denotes an anonymous variable the value of which is not used subsequently.)

The functions *push, pop* and *top* are used as the usual operations on stacks. The plus sign (+) denotes the addition on natural numbers. The subroutine call chain is implemented as a sequence of labels with ε denoting the empty sequence, the dot ($\cdot$) operator used as adding a label to a sequence. Later on, we will use the predicate $\in$, $a \in u$ to be true iff the label a occurs in the sequence u. *split* is a function which splits a sequence into two parts at a location pointed to by the first occurrence of a given value (suppose $a \notin u$):

$$split(uav,\ a) = (ua,v).$$

All these function are *external* (or *static*) w.r.t. the ASM. This means, they are not updated by ASM rules and are not part of the dynamic aspects of the ASM. To build a finite abstraction of the ASM, it is required to give finite abstractions of these functions.

2.3 Concrete Types and their Abstraction

In order to create a finite state system from a bytecode program it is necessary to restrict the universes used in the ASM rules to finite sets and to give finite interpretations of the used functions. By providing these abstractions, we define the corresponding abstract rules which can be composed to build a ASM rule that describes a finite model of the bytecode program.

First, we describe the used concrete types. For a given bytecode program, the number of used variables is fixed. They are numbered onwards from 0, so their index range *Varnum* is an interval of the natural numbers starting from 0. *Nat* denotes the natural numbers. The universe *Word* contains all values of primitive types. The type identifiers are given in the left column, and the state components (internal ASM functions) in the right one:

$$\begin{array}{ll} Pc == Nat & pc : Pc \\ PrimType == Word & \\ Loc == Varnum \rightarrow PrimType & loc : Loc \\ Opd == Stack(PrimType) & opd : Opd \\ Sr == Nat^{*} & sr : Sr\ . \end{array}$$

Now, we give finite abstractions of the types in the left column. Their structure is generic but their size depends on the given bytecode program. However, they are fixed for and can be computed from a given bytecode program. Also, the maximum height h of the operand stack is fixed for a given bytecode program (and supplied in the header of the bytecode program). Suppose the bytecode instructions are given in an array *code* with the length *codelength*.

$$\begin{array}{l} Pc == \{1,\ldots,codelength\} \cup \{undef\} \\ PrimType == Pc \\ Loc == Varnum \rightarrow PrimType \\ Opd == Stack_h(PrimType) \\ Lab == \{L : ex.\ i\ with\ code(i) = jsr\ L\} \\ Sr == Lab^{*} \cup \{undef\} \end{array}$$

Pc is explicitly restricted to the possible addresses of instructions in the *code* array, extended by a special value *undef* denoting an invalid address. *PrimType* is restricted to instruction addresses as the instructions we consider here use this type only. *Lab* substitutes the natural numbers *Nat* in *Sr*. These universes are sufficient for our purpose, as we consider a very limited set of instructions only. However, they can be extended if more instructions are considered.

In ASMs, the value *undef* is actually contained in any universe by default. However, we include this value explicitly to make clear which values are used and that they must be properly included later in the description of the finite state system.

The structure of the instruction rules is not changed when building a finite ASM. But the external functions are substituted with finite abstractions. We give the definitions of the abstracted functions:

$$push(u,a) = \begin{cases} \{au\} & \text{if } |u| < h \\ \{undef\} & \text{otherwise} \end{cases}$$

$$pop(u) = \begin{cases} \{v\} & \text{for } u = av \\ \{undef\} & \text{if } u \in \{\varepsilon, undef\} \end{cases}$$

$$top(u) = \begin{cases} \{a\} & \text{for } u = av \\ \{undef\} & \text{if } u \in \{\varepsilon, undef\} \end{cases}$$

$$n + i = \begin{cases} \{m\} & \text{with } m = n{+}i, \text{ if } n{+}i \leq codelength \\ \{undef\} & \text{otherwise} \end{cases}$$

$$a \cdot u = \begin{cases} \{au\} & \text{if } a \notin u \\ \{undef\} & \text{otherwise} \end{cases}$$

$$split(u,a) = \begin{cases} \{(wa,v)\} & \text{if } u = wav \\ \{undef\} & \text{otherwise} \end{cases}$$

By substituting these definitions in the ASM rule for a bytecode program, we get a *finite ASM* since all universes are finite and no universe extensions occur. This ASM can be seen as a description of a *finite state system* where the state space is defined by the possible values of the (internal) functions.

Note that in the general case, the abstract functions are *nondeterministic* (though there are no nondeterministic functions for the subset of bytecode instructions we consider here). Therefore, the right hand sides of the defining equations are sets of possible values. During "execution" of the ASM rule, one of the possible values is nondeterministically chosen to determine the successor state.

Note also that the abstracted functions have to be *homomorphic* w.r.t. to the concrete functions, i.e. the condition

$$y = f(x) \Rightarrow h(y) \in f_h(h(x))$$

must hold for a mapping h from concrete to abstract domains. This ensures that conditions that are proven to hold in the abstract system carry over to the concrete system. This applies only to CTL formulae that are quantified over all execution paths, therefore we will forbid the use of the `E` quantifier (stating the existence of a path) in specifying bytecode properties. (See [3] for further details.)

3 Building a Finite State System

Our approach is based on unlabelled transitions, since a bytecode program takes no input during execution. The choice of the successor state depends only on the internal state of the program.

A bytecode program is executed deterministically by the VM. By abstracting to finite types, conditional branches become nondeterministic choices since the branching depends on concrete values that are not accessible anymore. Therefore, a successor state is nondeterministically chosen in the abstract system from the set of possible successor states which is determined by the transition relation. Thus, we define finite state systems as follows.

A *finite state system* is a tuple $M = (S, R, I)$ where S is a finite set of states; $R \subseteq S \times S$ is the transition relation; $I \subseteq S$ is the set of initial states.

3.1 Translation of ASM Rules

For model checking, we are interested in a representation of the state system on which the necessary computations can be performed efficiently. The model checker SMV, for example, takes a propositional formula as an input description of the transition relation where two variable sets X and X' occur. The variables $x \in X$ denote the *current* values of the state variables and $x' \in X'$ denote the *successor* values. By internally representing a propositional formula as a BDD, the transition relation can be concisely represented and the Boolean operations can be computed efficiently.

The state space S consists of the ASM (nullary) functions *pc, loc, opd, sr*. For SMV, these translate to the following state variables:

`pc`	*pc*
`loc0,...,loc`N	for the array *loc*
`opd0,...,opd`H_1	for the stack positions on *opd*
`opdsize`	holding the current size of *opd*
`sr0,...,sr`H_2	for the positions in the list *sr*
`srsize`	holding the current size of *sr* .

(Note that this is just a suggestion how the state space can be represented in SMV. There are many possibilities to translate a finite ASM into a SMV model.)

The initial state is determined as follows. `pc` is set to 1. The `loc`$_i$ variables are set to initial values according to the signature of the bytecode program. All `opd`$_i$ and `sr`$_j$ are initialized to *undef* and `opdsize` and `srsize` are set to 0. This matches the initializations made by the JVM when starting to execute a bytecode program.

To feed our transition relation (represented by an ASM rule) into a model checker, we have to translate it into a propositional formula ϕ. The transition relation R_ϕ represented by this formula must fulfill the condition

$$R_\phi = \{(s,t) : \text{the rule fires at } s \text{ and results in } t\}.$$

(An efficient algorithm to construct a representation of R_ϕ from a given rule is currently being implemented.)

3.2 Propositions

State properties are described in terms of propositional formulae built from atomic propositions. Using SMV, atomic propositions are essentially equations or integer comparisons. All common propositional connectives and the CTL-quantifiers `A, G, F` may be used to build propositions. However, we require that negations occur only at the level of state propositions (i.e., formulae without any CTL-quantifiers) to be able to carry over properties of the abstract system to the concrete one (see [3]).

The propositions we consider in the next section use operations which are not allowed in SMV syntax. These operations have to be translated to propositional formulae in the same manner as in the ASM rule describing the transitions.

4 Security Properties

The constraints on bytecode programs are informally described in [6]. We give a formalization in terms of CTL formulae and argue that they meet the informal description. One interesting task would be to validate our formulae against other descriptions like [4, 9, 11].

The constraints in question are in fact *safety* properties since they refer to single program points only. Therefore, each constraint can be formulated as

`AG pc =` n `->` ϕ .

The preceding quantifier `AG` says that the following formula must hold on every state in every execution path. The condition `pc =` n restricts the checking of ϕ to those states that represent the program point n.

One of the general constraints on bytecode programs is that there are neither underflows nor overflows on the operand stack at any program point. This condition can be expressed by the formula

`AG` $0 \leq$ `opdsize` $\leq H_1$.

Another condition is the correct typing of instruction parameters, i.e. operand stack locations or local variables. As an example, consider the instruction *iload_0* which assumes an integer value in the local variable indexed by zero. For each address n where such an instruction occurs, we add the following formula to the specification:

`AG pc =` n `-> loc0 =` *int* ,

where *int* is the ecoding for an arbitrary integer value.

4.1 No Recursion Allowed

It must be assured that no subroutine is recursively called. As we are recording the active subroutines in the sequence *sr*, this constraint can be expressed as the following pre-condition of the `jsr` instruction:

$L \notin sr$

Thus, for each program point n with *code(n)* = *jsr* L we add the following formula to the model's specification:

`AG pc =` n `->` $L \notin sr$

4.2 Returning from Subroutines

Another constraint is that whenever a *ret* x instruction is executed, x must contain a return address value and the respective subroutine must be active. This is expressed as a pre-condition to the `ret` instruction:

$\varphi \equiv \bigvee_{(r,L)\in Z} \ (x = r \rightarrow L \in sr)$

where $Z = \{(r, L) : code(r - 1) = jsr\ L,\ r > 1\}$ which is the set of valid return addresses associated with the respective subroutine entry labels. This formula expands to a SMV formula φ' in a straightforward way.

The CTL formula we have to check for each program point n where a `ret` instruction occurs then is:

`AG pc =` n `->` φ'

5 Related and Future Work

Our approach is based on a formal semantics of Java bytecode. Our description builds mainly upon the original description in [6] but we are aware of the existing formalizations in [1, 13, 2, 9].

Some notations were used from [2] where Abstract State Machines are used to describe the Java Virtual Machine with the goal of defining a platform for correct compilation of Java code.

It remains to connect our approach to other formalizations of bytecode verification, e.g. the one in [4]. This could lead to higher confidence in our approach and reveal if there are flaws in any of the formalizations.

By providing a generic tool which takes an abstract interpretation of bytecode semantics and formulae describing system properties, we hope to extend bytecode verification to check further properties of bytecode programs. One candidate, especially important in the field of Java smart cards, could be the resource consumption of bytecode programs. One would have to take into cosideration certain garbage collection strategies used with smart cards to check that a bytecode program leaves no objects on the heap and give an appropriate abstract interpretation for the bytecode instructions and appropriate formulae.

References

1. Peter Bertelsen. Semantics of Java Byte Code. http://www.dina.kvl.dk/ ~pmb, March 1997.
2. Egon Börger and Wolfram Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In *23rd International Symposium on Mathematical Foundations of Computer Science*, LNCS. Springer-Verlag, 1998.
3. E. Clarke, D. Grumberg, and D. Long. Model Checking and Abstraction. *ACM Trans. on Prog. Languages and Systems*, 16(5):1512–1542, 1994.
4. Allen Goldberg. A Specification of Java Loading and Bytecode Verification. http://www.kestrel.edu/HTML/people/goldberg/Bytecode.ps.gz, 1997.
5. Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.
6. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
7. Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
8. Joachim Posegga and Harald Vogt. Byte Code Verification for Java Smart Cards Based on Model Checking. In *5th European Symposium on Research in Computer Security*, LNCS. Springer-Verlag, 1998.
9. Z. Qian. A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer-Verlag, 1998.
10. Emin Gün Sirer, Arthur J. Gregory, and Brian N. Bershad. Kimera: A Java System Architecture. http://kimera.cs.washington.edu/, 1998.
11. Raymie Stata and Martín Abadi. A Type System for Java Bytecode Subroutines. In *Proc. 25th ACM Symp. Principles of Programming Languages*. ACM Press, 1998.
12. Bernhard Steffen. Data Flow Analysis as Model Checking. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 346–364. Springer-Verlag, September 1991.
13. Wolf Zimmermann and Thilo Gaul. An Abstract State Machine for Java Byte Code, June 1997.