Bytecode Verification by Model Checking

David Basin, Stefan Friedrich and Marek Gawkowski
({basin,friedric,gawkowsk}@informatik.uni-freiburg.de)
Albert-Ludwigs-Universität Freiburg

Abstract. Java bytecode verification is traditionally performed using dataflow analysis. We investigate an alternative based on reducing bytecode verification to model checking. First, we analyze the complexity and scalability of this approach. We show experimentally that, despite an exponential worst-case time complexity, model checking type-correct bytecode using an explicit-state on-the-fly model checker is feasible in practice, and we give a theoretical account why this is the case. Second, we formalize our approach using Isabelle/HOL and prove its correctness. In doing so we build on the formalization of the Java Virtual Machine and dataflow analysis framework of Pusch and Nipkow and extend it to a more general framework for reasoning about model-checking based analysis. Overall, our work constitutes the first comprehensive investigation of the theory and practice of bytecode verification by model checking.

Keywords: Static analysis, model checking, type safety, bytecode verification

1. Introduction

1.1. Bytecode verification and model checking

Java is a popular programming language, well-suited for building distributed applications where users can download and locally execute programs. To combat the security risks associated with mobile code, Sun has developed a security model for Java in which a central role is played by bytecode verification [15, 31], which ensures that no malicious programs are 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. The essential, and non-trivial, part of bytecode verification is checking type-safety properties of the bytecode, i.e., that operands are always applied to arguments of the appropriate type and that there can be no stack overflows or underflows. By checking these properties statically prior to execution, the JVM can safely omit the corresponding runtime checks.

In this paper we investigate an alternative approach to bytecode verification: the use of model checking to validate the type-safety properties of Java bytecode programs. We explain how this approach works, analyze theoretically and experimentally when it is feasible and how it scales, and formally state and prove its correctness. In doing so, we give the first

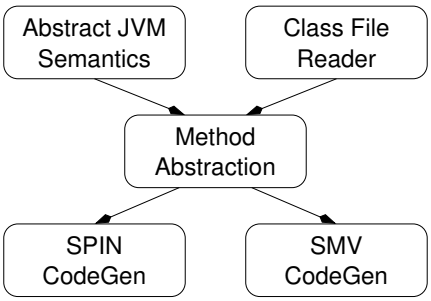


Figure 1. System architecture

comprehensive account of the theory and practice of bytecode verification based on model checking.

Our account is based on two developments. The first is a system for bytecode verification, whose module structure is depicted in Figure 1. The class file reader parses a Java class file, extracting its methods. The methods, together with the abstract semantics of the Java Virtual Machine, are passed to a method abstraction module, which produces an abstract, intermediate representation of each method. This representation consists of a transition system (describing the method's execution on the abstract machine) and additionally a specification of safety properties that formalize conditions sufficient for the bytecode's type safety. These descriptions are translated by code generators into the input language of the SPIN [10] and SMV [16] model checkers.

Using our system, we carry out experiments that show that despite an exponential worst-case time complexity, model checking type-correct bytecode is feasible in practice when carried out using an explicit-state, on-the-fly model checker like SPIN. The situation here is similar to type checking in functional programming languages like ML, where the typability problem for terms is DEXPTIME hard [11], yet the worst-case complexity is not a problem in practice. In addition, we investigate this theoretically and explain the practical advantages of the model-checking approach.

The second development is a formal theory in which we formalize and prove correct the model-checking approach to bytecode verification. Our theory is constructed using the Isabelle/HOL system, which is a formalization of higher-order logic within the Isabelle theorem prover [19, 20]. To accomplish this task, we build upon the Isabelle/HOL formalizations of the JVM [22, 23] and the abstract verification framework that Nipkow developed for verifying dataflow algorithms for bytecode verification [17]. This framework formalizes the notion of a well-typing for bytecode programs and proves that a bytecode verifier is correct (i.e., accepts only

programs free of runtime type errors) when it only accepts programs possessing a well-typing (formalized in Section 6.3). We extend this framework to support polyvariant dataflow analysis and model checking. Using this extension, we prove that every bytecode program whose abstraction globally satisfies the instruction applicability conditions (which is what is established by model checking) possesses such a well-typing, i.e., we validate the model-checking approach by proving a theorem roughly of the form

$$(\mathit{abstraction}(\mathit{Method}) \models_{LTL} \Box\, \mathit{app\_conditions}(\mathit{Method})) \\ \Longrightarrow (\exists\, \phi.\ \mathit{well\_typing}(\phi, \mathit{Method}))\,.$$

1.2. Contributions

We believe that there are several reasons why this work should be of interest to other researchers. First, we show how to use formal methods to improve the security and precision of the entire bytecode verification and execution process. Our implementation declaratively formalizes both the abstract semantics of the JVM and how to generate type correctness properties as temporal safety properties. These formalizations, given in the ML programming language, are used directly to produce the transition system and the type correctness properties that are verified by the model checker. 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.

Our formal theory provides the explicit link to type safety via the existence of well-typings. By making this model explicit and precise, as well as the reasons for its correctness, our formalization goes far beyond the Java documentation [15], 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.

Bytecode verification by model checking is also more complete than the conventional approach. As Stärk and Schmid point out [29], there are Java programs whose compiled bytecode is type-correct that 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 show that the model-checking approach is practical and scales to realistic examples. This 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 and space complexity that is exponential in the worst case.

Our experiments show that for type-correct bytecode, model checking can be performed in a tractable way, using the SPIN model checker, which

constructs the state space incrementally, on-the-fly. This thesis was tested on the 10388 methods of the `java` library distributed with Sun's Java Development Kit, where 10318 (which is 99.3%) are verified in under a second. The reason is that although there are exponentially many states, for correct code arising in practice, only polynomially many states are usually reachable (see Section 3 for the exact analysis). This is in contrast to symbolic BDD-based model checkers like SMV, which 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 in reasonable time (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 (such as property simplification, as described in Section 3.3) to detect an error.

This result suggests the usefulness of bytecode verification by model checking, especially in application domains where time and space requirements are less important than correctness and possible extensibility. One such application is Java for smart cards (JavaCard), a 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 runtime environment. One of our original motivations for this work was to investigate whether model checking could be used as an alternative in this domain, along the lines suggested in [21]. We can now answer this question positively.

Third, our correctness results also help to clarify the general relationship between bytecode verification by dataflow analysis and bytecode verification by model checking. These two directions are related: both are based on abstract interpretation and solving fixpoint equations. However, while dataflow analysis computes a type for a method and checks that the method's instructions are correctly applied to data of that type, the model-checking approach is more declarative; here one formalizes the instruction applicability conditions as formulae in a temporal logic (e.g. LTL) and uses a model checker to verify that an abstraction of the method (corresponding to the abstract interpreter of the dataflow approach) satisfies these applicability conditions. The Isabelle/HOL development, which we present in Sections 4–7, also provides insights into the relationship between monovariant and polyvariant dataflow analysis and model checking, in particular, what differences are required in their formalization.

In addition, our correctness results have some generality. Since we formalize model checking declaratively, not algorithmically, as done in [27], our Isabelle/HOL formalization makes a general statement about the correctness of the model-checking approach. This statement is indepen-

dent of the implemented model-checking algorithm and can be applied, for example, to reason about the correctness results of different model checkers like SPIN and SMV.

There is a final reason why others may be interested in this work: our approach gives rise to an unlimited supply of scalable, real-life model-checking problems. 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 test and compare different model checkers. Our system is freely available for such benchmarking purposes.

1.3. Related work

In the recent years, there has been a convergence of ideas in static analysis and model checking. Namely, different kinds of program analysis can be performed by fixpoint computations and these computations can either be carried out by specialized algorithms or by general purpose model checkers [27, 28]. Moreover, both static analysis and model checking generally reason about abstractions of programs, e.g., abstracting the operational model by identifying data.

While static analysis techniques have a long 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 [21], who carried out a few small examples by hand to suggest how, in principle, this approach could work for a subset of the JVM. This was the starting point for the development of our system and together with Posegga and Vogt we built our first prototype verifier [3] for model checking bytecode programs based on a subset of the JVM. The system reported on here represents a further development of these ideas (superseding the preliminary work reported on in [1, 2]) and our experiments are the first large-scale effort to apply model checking to bytecode verification and to study its practical significance.

The Java language and the JVM have both been the focus of numerous formal studies. This activity has been motivated by the widespread use of Java, the lack of formal treatment originally given in [15], and the interplay between different kinds of analysis and security, both in the sense of type safety and IT security. A number of different approaches have been proposed for type checking bytecode and an excellent overview of the area is provided in [14]. Most of this work is theoretically oriented and is concerned with formalizing models of the JVM [7, 8, 24] and defining related type systems [6, 9, 25, 30].

Most relevant to our work is the research on formally proving the soundness of various approaches to bytecode verification or verifying suf-

ficient conditions for bytecode verifiers to be correct [4, 7, 12, 13, 17, 18, 22, 23]. As we will explain in detail in Section 5, our formal theory builds upon the work and theories of Pusch [23], Nipkow [17], and Klein [12, 13] who formalized a model of the JVM in Isabelle/HOL as well as theories for verifying bytecode verifiers based on dataflow algorithms.

1.4. Organization

The remainder of this paper is organized as follows. In Section 2 we explain how to abstract class files to model-checking problems. This provides the basis for the experiments and analysis that we report on in Section 3. In Section 4 we explain how we build upon the work of Pusch and Nipkow to formalize the correctness of the model-checking approach in Isabelle/HOL. We present background theories in Section 5, our adaptation of the JVM model in Section 6, and our proof of correctness in Section 7. Finally, we draw conclusions in Section 8.

2. Abstracting class files 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.

2.1.1. *Bytecode and the* JVM

Java programs are compiled to bytecode instructions that are interpreted by the JVM. The result of compilation, a class file, contains a symbol table (called the constant pool) describing the fields of the class and a list of the methods of the class. Figure 2 provides an example, which will be used throughout this section: a Java implementation of lists and the bytecode of the method `Cons.length()`, which is compiled from the corresponding Java method. The first instruction of this bytecode method loads the `this` reference, which is stored in the local variable 0, on the operand stack. Using this reference, the reference to the tail of the list is then fetched from the field `tl` and pushed on the operand stack by the `getfield` instruction, and the `length()` method is (recursively) invoked by the `invokevirtual` instruction. Next, the integer constant 1 is pushed on the operand stack and added to the result of the method invocation. Finally, the resulting integer is returned by the `ireturn` instruction.

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

```
public abstract class EmptyListException extends Error {}

public abstract class List {
 public static final List nil = new Nil();
 public List cons(Object hd) {return new Cons(hd, this);}

 public abstract Object head();
 public abstract List tail();
 public abstract int length();
}

class Nil extends List {
 public Object head() {throw new EmptyListException();}
 public List tail()   {throw new EmptyListException();}
 public int length()  {return 0;}
}

class Cons extends List {
 Object hd;
 List   tl;

 public Object head() {return hd;}
 public List tail()   {return tl;}
 public int length()  {return tl.length() + 1;}
}



Method int Cons.length()
max_stack=2, max_locals=1
 0 aload_0
 1 getfield Cons.tl
 2 invokevirtual List.length()
 3 iconst_1
 4 iadd
 5 ireturn
```

Figure 2. Java implementation of lists and bytecode of method `Cons.length()`

stack, which is used to evaluate expressions. For instance, the `iadd` instruction adds the two topmost elements of the operand stack, discards those elements from the stack, and pushes the result of the addition back on the stack. In addition to the operand stack, the JVM also uses an array of registers to store local variables, e.g., the local variable 0 used in the `aload` instruction.

Most JVM instructions are typed. For instance, the `getfield Cons.tl` instruction of the method `length` (see Figure 2), which accesses the `tl` field in class `Cons`, requires that the operand stack contains a reference to an object of class `Cons` (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.

2.1.2. *Bytecode verification*
To guarantee the secure operation of the JVM, one must show that each method is well-typed. This means that one can assign a state type to each point in the program, where 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 $(\mathsf{Empty}, loc[0 \mapsto \mathsf{REF}(Cons)])$ associates to a program point those states whose operand stack is empty and whose first local variable contains an object of class *Cons*. 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.

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

The type of a program point is constructed by computing the supremum (see Section 5 for a formal definition) of all state types that the program point assumes when reached on different execution paths. This requires the existence of a unique supremum, which is not the case for the JVM due to multiple inheritance of interfaces. There are several solutions to this problem, including considering sets of types instead of single types or leaving the checks for the correct implementation of interfaces to the Virtual Machine at runtime (as is 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 are incompatible. Solutions include structural restrictions on bytecode (Sun's approach), the use of type sets instead of mere types [6], and polyvariant dataflow analysis [14]. Polyvariant analysis associates to a program point a contour-indexed family of state types, where a contour approximates the control flow leading to a program point; for bytecode verification, call-stacks containing the return addresses of (nested) subroutines, i.e. the program points after the `jsr` instructions, represent the contours. An instruction is then analyzed once on each contour. In contrast, monovariant analysis associates only one program state type to each program point (which corresponds to polyvariant analysis with empty contours). It is an open question as to which solution is best. The polyvariant approach is more elegant but its time complexity is exponential in the depth of subroutine nesting.

2.1.4. *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 on the states of the abstract transition system and use off-the-shelf model checkers to determine whether these properties are satisfied. The model checker then either reports that the method is correct, or it provides a counterexample in the form of an execution trace that leads to a type error.

This approach is simpler than conventional bytecode verification as the correctness requirements can be clearly and comprehensibly specified (see Section 2.3), which helps to avoid errors in the formalization. Moreover, multiple inheritance is unproblematic since the formalization of the correctness properties only requires that the types are partially ordered.

Furthermore, note that the problems with subroutines, alluded to above, do not arise, as model checking performs, in the words of Leroy [14, p. 281], "the ultimate polyvariant analysis." Namely, model checking associates to a program point one state type per execution path leading to that program point. This entails that model checking is polyvariant also with respect to branching instructions, and thus is less restrictive than conventional monovariant analysis, as demonstrated by the following bytecode method.

```
Method int Example.m(int)
max_stack=2, max_locals=2
 0 iload_1   // load int parameter on stack
 1 ifeq +3   // if top of stack is 0, branch to line 4
 2 iload_1   // again load int parameter on stack
```

```
3 goto 5    // at line 5 the different paths merge
4 aload_0   // load this reference on stack
5 iload_1   // load int parameter on top of stack
6 ireturn   // return the top of stack
```

In this method, the `iload` instruction at line 5 can be reached via two different execution paths. On the first path an integer is loaded on the operand stack (line 2) whereas on the second path the `this` reference of the called object is loaded (line 4). With conventional bytecode verification, this method would be rejected since the primitive type INT and the reference type are incompatible and therefore the state types belonging to the different paths cannot be merged. However, under our model checking approach, the method is accepted as well-typed since for all instructions the applicability conditions are fulfilled.

The downside of the more precise analysis is, as in the case of polyvariant dataflow analysis, a time complexity exponential in the number of variables used (the number of local variables and the size of the operand stack). We investigate this problem and its implications in Section 3.2.

2.2. Abstract transition system

We abstract a method M to a finite state transition system (Q, q_0, Δ). The set of states $Q \subseteq \mathbb{N} \times (\mathcal{T}\ \mathsf{stack}) \times (\mathcal{T}\ \mathsf{array})$ 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 (NULLT represents the polymorphic type of the null reference) and the program addresses (since they can be targets of `ret` instructions). We add an element ERR to represent uninitialized values.

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

Only a finite subset of $\mathcal{T}$ actually occurs in a particular transition system, and we can compute this subset by inspecting the signature and the instructions used in the method M. The signature of M, which has the form

$$S = (\mathit{mname}, \mathit{result\_type}, [\mathit{arg\_type}_1, \ldots, \mathit{arg\_type}_n]),$$

specifies the method's name, its result type, and its argument types. The method body is given as a list *ins* of bytecode instructions, where ins_p refers to the pth element of this list. The types introduced by the

instructions of the method body are then determined as follows (we list only a few examples).

$$\begin{array}{lcl}
\mathsf{types_of_instr}(\texttt{iadd}) & = & \{\mathsf{INT}\} \\
\mathsf{types_of_instr}(\texttt{new}\ C) & = & \{\mathsf{REF}(C)\} \\
\mathsf{types_of_instr}(\texttt{getfield}\ C.f) & = & \{\mathsf{REF}(C), \tau\}, \\
 & & \text{where } \tau \text{ is the type of field } C.f
\end{array}$$

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

$$\begin{array}{rcl}
T & = & \{\mathsf{REF}(C), \mathit{result_type}, \mathit{arg_type}_1, \ldots, \mathit{arg_type}_n\} \cup \\
 & & \bigcup_{p \in \{0,\ldots,|\mathit{ins}|-1\}} \mathsf{types_of_instr}(\mathit{ins}_p)\,.
\end{array}$$

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 n parameter instances, which are passed through the first $n+1$ local variables. The remaining local variables, $\mathit{loc}[n+1], \ldots, \mathit{loc}[\mathit{max\_locals}]$, are initially undefined ($\mathit{max\_locals}$ is specified in the class file), i.e., $q_0 = (0, \mathsf{Empty}, \mathit{loc})$, where

$$\begin{array}{rcl}
\mathit{loc}[0] & = & \mathsf{REF}(C) \\
\mathit{loc}[1] & = & \mathit{arg_type}_1, \ldots, \mathit{loc}[n] = \mathit{arg_type}_n \\
\mathit{loc}[n+1] & = & \mathsf{ERR}, \ldots, \mathit{loc}[\mathit{max_locals}] = \mathsf{ERR}\,.
\end{array}$$

The transition relation is defined as $\Delta = \bigcup_{p \in \{0,\ldots,|\mathit{ins}|-1\}} \{(q, q') \mid q' \in \mathsf{abs\_instr}\ \mathit{ins}_p\ q\}$, where the function $\mathsf{abs\_instr}$ maps an instruction i and a state q to the set of q's successor states. We give a few representative examples here, which show how the state q, consisting of the program counter pc, the operand stack opst, and the local variables loc, is modified.

$$\mathsf{abs\_instr}\ i\ (\mathit{pc}, \mathit{opst}, \mathit{loc}) =$$

$$\begin{cases}
\{(\mathit{pc}+1, \mathit{loc}[n].\mathit{opst}, \mathit{loc})\}, & \text{if } i = \texttt{iload}\ n \\
\{(\mathit{pc}+1, \mathsf{pop}(\mathit{opst}), \mathit{loc}[n \mapsto \mathsf{top}(\mathit{opst})])\}, & \text{if } i = \texttt{istore}\ n \\
\{(\mathit{pc}+1, \mathsf{INT}.\mathsf{pop}(\mathsf{pop}(\mathit{opst})), \mathit{loc})\}, & \text{if } i = \texttt{iadd} \\
\{(\mathit{pc}+1, \tau.\mathsf{pop}(\mathit{opst}), \mathit{loc})\}, & \text{if } i = \texttt{getfield}\ C.f \text{ and} \\
 & \tau \text{ is the type of field } f \\
\{(\mathit{pc}+1, \mathsf{pop}(\mathit{opst}), \mathit{loc}), & \text{if } i = \texttt{ifeq}\ \mathit{offset} \\
\ (\mathit{pc}+\mathit{offset}, \mathsf{pop}(\mathit{opst}), \mathit{loc})\}, & \\
\{(\mathit{pc}+\mathit{offset}, \mathsf{ADR}(\mathit{pc}+1).\mathit{opst}, \mathit{loc})\}, & \text{if } i = \texttt{jsr}\ \mathit{offset} \\
\{(\mathsf{retaddr}(\mathit{loc}[n]), \mathit{opst}, \mathit{loc})\}, & \text{if } i = \texttt{ret}\ n \text{ and} \\
 & \mathsf{retaddr}(\mathsf{ADR}(p)) = p \text{ for all } p \\
\{(\mathit{pc}, \mathit{opst}, \mathit{loc})\}, & \text{if } i = \texttt{ireturn}
\end{cases}$$

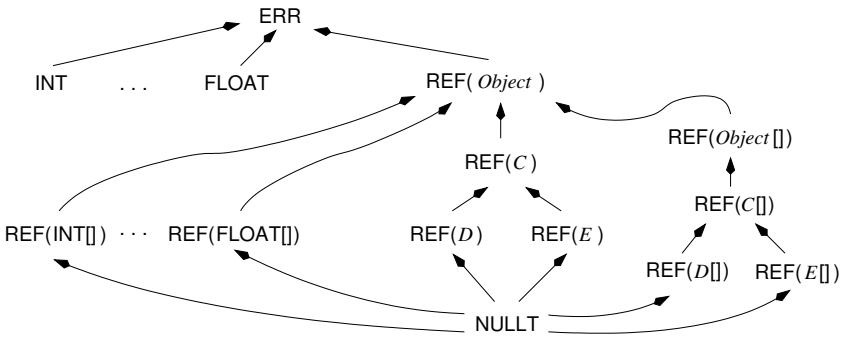


Figure 3. Subtyping relation $\sqsubseteq_P$

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 max_stack can be computed by the Java compiler in advance. This value is given as a method attribute in the class file of the method. We can formulate the corresponding condition as $\mathsf{size}(opst) \leq max\_stack$.

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 $\sqsubseteq_P$, which depends on the program P. Figure 3 shows the subtyping relation $\sqsubseteq_P$ for a program P consisting of three classes C, D, and E, where D and E are each a direct subclass of C. In the specification of the JVM, the applicability conditions can be formalized in a straightforward, declarative way. We give here a few representative examples.

`iload` n	$\mapsto$	$loc[n] = \mathsf{INT}$
`istore` n	$\mapsto$	$\mathsf{top}(opst) = \mathsf{INT}$
`iadd`	$\mapsto$	$(\mathsf{top}(opst) = \mathsf{INT}) \ \wedge \ (\mathsf{top}(\mathsf{pop}(opst)) = \mathsf{INT})$
`getfield` $C.f$	$\mapsto$	$\mathsf{top}(opst) \sqsubseteq_P \mathsf{REF}(C)$
`ifeq` *offset*	$\mapsto$	$(\mathsf{top}(opst) = \mathsf{INT})$
`jsr` *offset*	$\mapsto$	*True*
`ret` n	$\mapsto$	$loc[n] \in adr$
`ireturn`	$\mapsto$	$\mathsf{top}(opst) = \mathsf{INT}$

For a given program P, the relation $\sqsubseteq_P$ is finite and thus the conditions can be automatically unfolded and checked. The condition for the

`getfield` $C.f$ instruction, for instance, would be unfolded to

$$\begin{aligned}\mathsf{app\_instr}(\texttt{getfield}\ C.f)\ (pc, opst, loc)\ \equiv\\ \mathsf{top}(opst) = \mathsf{REF}(C)\ \vee\ \mathsf{top}(opst) = \mathsf{REF}(D)\ \vee\\ \mathsf{top}(opst) = \mathsf{REF}(E)\ \vee\ \mathsf{top}(opst) = \mathsf{NULLT}.\end{aligned}$$

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.

$$(\mathsf{size}(opst) \leq max\_stack) \wedge \bigwedge_{p \in \{0,\ldots,|ins|-1\}} (pc = p \Rightarrow \mathsf{app\_instr}(ins_p))$$

2.4. Backends for SPIN and SMV

We have implemented two backends: one for SPIN and one for SMV. The idea behind 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 should 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 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 `Cons.length()` bytecode presented in Section 1. The data required to model this method are the address labels $\{\mathsf{ADR}(0), \ldots, \mathsf{ADR}(5)\}$ and the types $\{\mathsf{ERR}, \mathsf{INT}, \mathsf{NULLT}, \mathsf{REF}(Cons), \mathsf{REF}(List)\}$. Initially the operand stack is empty and the local variable $loc[0]$ contains the *this* reference $\mathsf{REF}(Cons)$. Each transition modeling an instruction is carried out as an atomic step.

In SPIN, the invariant φ can be expressed in different ways, e.g., as an observer process or using a SPIN never-claim. We have chosen the former: The observer process runs interleaved with the abstract state transition system and it checks that the assertion, which states the correctness property, holds at each state. This approach is a simple and efficient way to formalize an invariance property in Promela [26] and has the practical advantage that temporal formulae need not be translated separately to automata. Figure 5 shows the correctness properties for our sample bytecode. For example, for the recursive call by the `invokevirtual List.length()` instruction at $pc = 2$, we require that

```
#define ADR0 0
#define ADR1 1
#define ADR2 2
#define ADR3 3
#define ADR4 4
#define ADR5 5
#define ERR  6
#define INT  7
#define NULLT 8
#define REF_CONS 9
#define REF_LIST 10

init {
  atomic { loc[0] = REF_CONS; opst_ptr = 0; pc = 0 };
  run assertions ();  run transitions ()
}

proctype transitions( ) {
 do
 :: pc==0 -> atomic { opst[opst_ptr]=loc[0];
                      opst_ptr=opst_ptr+1;
                      pc=1 };

 :: pc==1 -> atomic { opst[opst_ptr-1]=REF_LIST;
                      pc=2 };

 :: pc==2 -> atomic { opst[opst_ptr-1]=INT;
                      pc=3 };

 :: pc==3 -> atomic { opst[opst_ptr]=INT;
                      opst_ptr=opst_ptr+1;
                      pc=4 };

 :: pc==4 -> atomic { opst[opst_ptr-2]=INT;
                      opst_ptr=opst_ptr-1;
                      pc=5 };

 :: pc==5 -> atomic { pc=pc }
 od
}
```

Figure 4. Abstract transition system for the `Cons.length()` bytecode in PROMELA

```
proctype assertions( ) {
 do
 :: assert (
     (pc!=0 || loc[0]==NULLT
            || loc[0]==REF_CONS
            || loc[0]==REF_LIST) &&
     (pc!=1 || opst[opst_ptr-1]==NULLT
            || opst[opst_ptr-1]==REF_CONS) &&
     (pc!=2 || opst[opst_ptr-1]==NULLT
            || opst[opst_ptr-1]==REF_CONS
            || opst[opst_ptr-1]==REF_LIST) &&
     (pc!=4 || opst[opst_ptr-1]==INT &&
               opst[opst_ptr-2]==INT) &&
     (pc!=5 || opst[opst_ptr-1]==INT) &&
     opst_ptr<=2 )
 od
}
```

Figure 5. Correctness properties for the `Cons.length() bytecode` in PROMELA

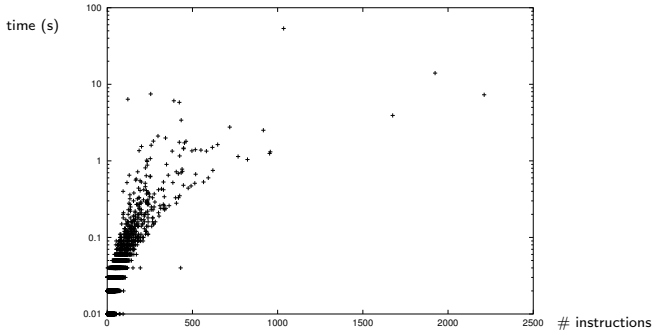
the topmost element of the operand stack is either the `null` reference (i.e. `opst[opst_ptr-1]==NULLT`) or a reference to an instance of the class `List` (i.e. `opst[opst_ptr-1]==REF_CONS` or `opst[opst_ptr-1]==REF_LIST`).

3. Experimental results and analysis

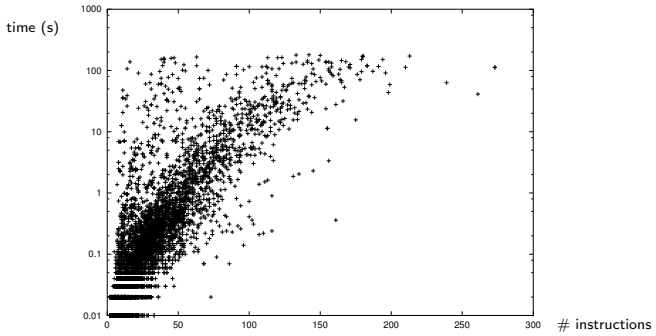
We have carried out two different kinds of experiments to investigate the applicability and scalability of model checking for bytecode verification. First, to test the practical applicability of this approach, we model checked all the methods associated with a large Java library, namely all methods of the `java` 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` package. This package contains 1242 classes with 10388 methods in total. These methods are representative for Java bytecode as they contain all the instructions of



(a) SPIN



(b) SMV

Figure 6. Verification times for the methods of the `java` library (log-scale)

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 are modestly sized (27.8 instructions per method on average). In this package, only 510 methods contain more than 100 instructions.

As Figure 6 indicates, SPIN checks almost all of these methods in negligible time.[1] For 69 methods, SPIN requires more than a second and 36 of them exceed the time bound of three minutes; each of these methods has more than 3000 reachable states. This performance is adequate for most applications and in particular is more than adequate for off-line verification (e.g., smart cards) and on-line verification for applications, such as web applets, where methods are generally small. The picture for SMV is completely different; even methods with less than 100 instructions can require more than two minutes to verify and, for 664 methods, SMV

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

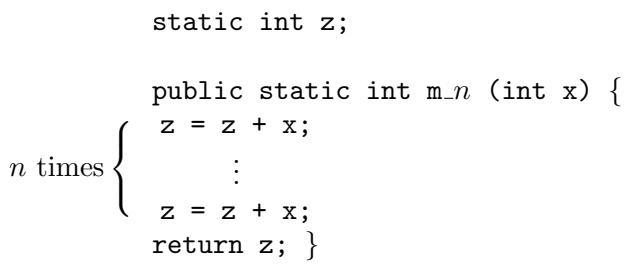


Figure 7. Method schema for testing complexity based on code length

exceeds the time bound of three minutes or runs 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 "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:

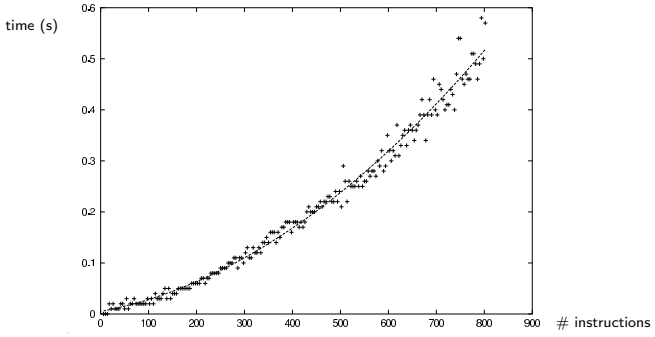
1. the length of methods, i.e. the number of instructions,
2. the number of local variables,
3. the depth of the class hierarchy, i.e. the number of types that are used to model a method, and
4. the depth of the nesting of subroutines.

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

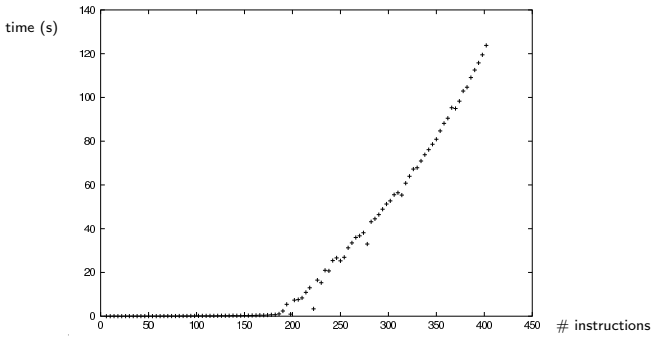
3.2.1. *Method length*

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



(a) SPIN



(b) SMV

Figure 8. Verification time depending on the number of instructions

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 using BDDs in SMV can lead to memory problems, even with good variable orderings, and results in runtimes orders of magnitude larger than SPIN's. It is interesting to see that these problems appear in even such a simple test.

3.2.2. *Number of local variables*
In this test, we varied the number of local variables, which also constitute the arguments of the method, from one to a maximum of 10. The method body of the ith method consists of a sequence of 10 `if`-statements (which keeps the length of the code constant) and uses i variables. In the first $11-i$ `if`-statements, only the variable `v1` is used, and each of the remaining $i-1$ `if`-statements uses a new variable. In the `then`-branch of an `if`-statement, the respective variable is assigned an object of class `A` and in the `else`-branch it is assigned an object of class `B`. This yields 2^i different possible assignments for the variables of method i. As bytecode

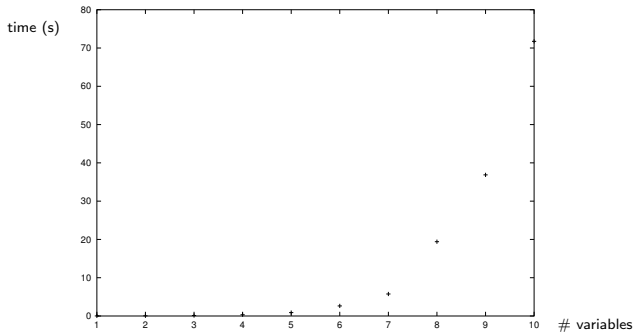
```
void m_1(int v0, Object v1)
  { if (v0 == 0) {v1 = new A(); } else {v1 = new B();}
    if (v0 == 0) {v1 = new A(); } else {v1 = new B();}
    if (v0 == 0) {v1 = new A(); } else {v1 = new B();} }

void m_2(int v0, Object v1, Object v2)
  { if (v0 == 0) {v1 = new A(); } else {v1 = new B();}
    if (v0 == 0) {v1 = new A(); } else {v1 = new B();}
    if (v0 == 0) {v2 = new A(); } else {v2 = new B();} }

void m_3(int v0, Object v1, Object v2, Object v3)
  { if (v0 == 0) {v1 = new A(); } else {v1 = new B();}
    if (v0 == 0) {v2 = new A(); } else {v2 = new B();}
    if (v0 == 0) {v3 = new A(); } else {v3 = new B();} }
```

Figure 9. Method schema testing complexity based on the number of local variables

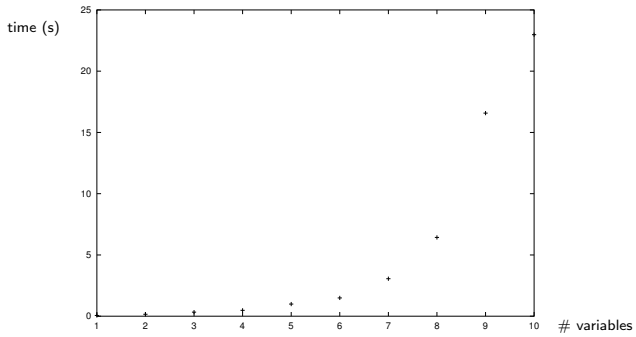
verification by model checking is polyvariant with respect to branches, all different assignments must be checked and this requires exponential time for both model checkers. Figure 9 shows an instance of our schema and Figure 10 displays the results.

3.2.3. *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. As shown in Figure 11, an additional class contains 100 methods, $m_1, \ldots, m_{100}$, to be checked. Each of these methods 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, the state space grows as more types are present; moreover, the properties checked also become more complex.

As Figure 12 shows, for 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). However, for 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 $i + 3$ bits to represent the types in this group and it appears that a small additional amount



(a) SPIN



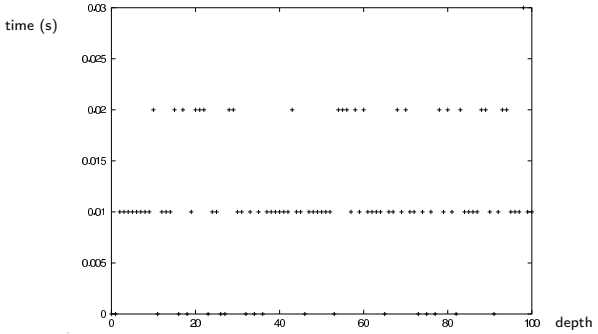
(b) SMV

Figure 10. Verification time versus number of local variables

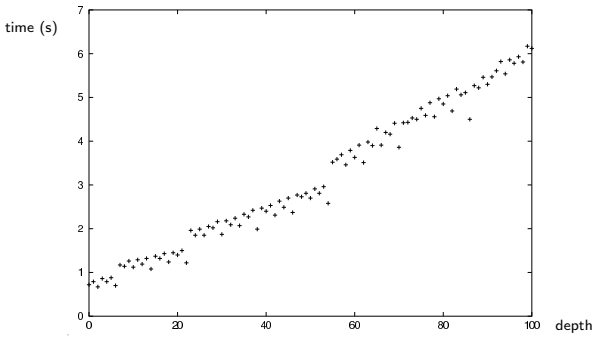
of time (corresponding to the small gap between the groups) is required to manipulate the larger BDDs. Note that the 100 local variables do not blow up the state space as only one of them, namely c_1, is actually used.

```
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;}
```

Figure 11. Method schema for testing complexity based on class hierarchy depth



(a) SPIN



(b) SMV

Figure 12. Verification time versus depth of class hierarchy

Hence, in this example, the BDDs succeed in exponentially compressing the state space.

3.2.4. *Depth of subroutine nesting*

In our last test, depicted in Figure 13, we examined methods that contain nested subroutines, where the nesting depth is increased in each method. To ensure that all compiled methods have the same number of instructions, we add an appropriate number of `x++` statements, which adjust the lengths of the methods.

As explained in Section 2.1, verifying subroutines is one of the more delicate issues in bytecode verification. The fact that verifying subroutines using model checking is much simpler than verifying them conventionally comes at the price of exponential time consumption! As Figure 14 illustrates, both SPIN and SMV perform poorly when checking nested subroutines. Since subroutines are polymorphic in the local variables that are not used in the subroutine, the reachable state space is exponential

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

  x++;                          // repeated 19*17 times
  try {throw new Exception();}
  catch (Exception e) {}
  finally {}
}

public void m_2() {
  int x = 0;

  x++;                          // repeated 18*17 times
  try {throw new Exception();}
  catch (Exception e) {}
  finally{
    try{throw new Exception();}
    catch (Exception e) {}
    finally{}
  }
}
```

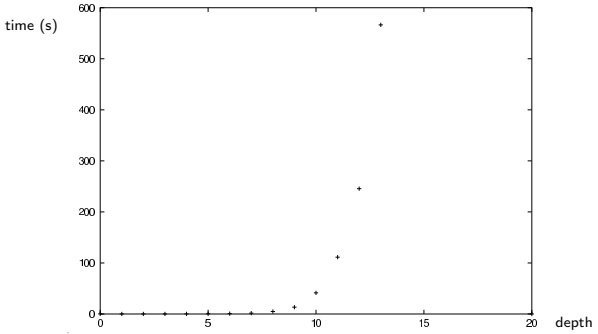
Figure 13. Sample methods for testing complexity of subroutine nesting

in the depth of subroutine nesting. BDDs do not significantly reduce this explosion.

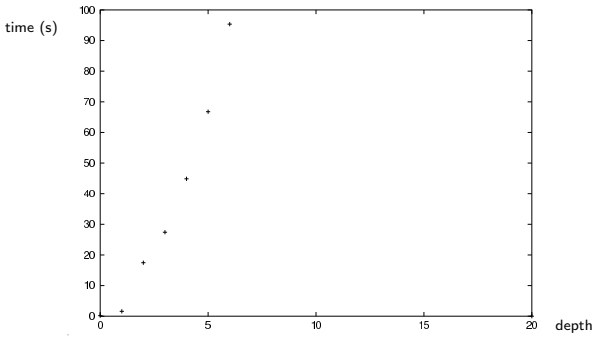
3.3. Analysis

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 of types, *max_locals* the number of local variables, and *max_stack* the maximal stack height. For conventional bytecode verification, where interfaces are treated as normal classes and correct typing of interfaces is checked at runtime, as is done by Sun's JVM, the size of the state space is $ins \cdot T^{max\_locals+max\_stack}$.

Despite the exponential 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



(a) SPIN



(b) SMV

Figure 14. Verification time versus depth of the subroutine nesting

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 fixpoint 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 storage locations (local variables and operand stack). Since no iteration can decrease the types associated with any program point, a fixpoint is reached in at most $ins \cdot T \cdot (max\_locals + max\_stack)$ iterations.

In the case of model checking, program points are also associated with types and, as in the dataflow analysis performed by Sun's verifier, the size of the state space is also $ins \cdot T^{max\_locals+max\_stack}$. However, since model checking is polyvariant with respect to branches, in the worst case all of these states are reachable (see Section 3.2.2).

The feasibility of model checking bytecode in practice depends on the algorithm used. Symbolic methods manipulate a representation of the entire state space and, as we have seen, often require exponential

resources to do so. However, in practice the nesting of subroutines has a fixed upper bound (it is almost never greater than two) and, after a branching instruction, variables are usually assigned values of the same types in both branches. In these cases, only polynomially many states are reachable, since there is only one state type possible for each program point. Since the property to be model checked is also linear in the number of instructions, it follows that an on-the-fly model checker like SPIN can validate correct bytecode in acceptable time and space.

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 finding errors is considerably more resource intensive than validating correct bytecode. Even for 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, which works for on-the-fly model checking. Instead of checking the correctness properties for all instructions simultaneously (e.g., the large conjunct in Figure 5), the properties to be 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 [5] is an interesting possible alternative, as normally the paths to errors are small.

4. Formalizing correctness

In the second half of this paper, we present a formal explanation of the correctness of our approach. Our formalization is within Isabelle/HOL and builds heavily upon the work of Pusch, Nipkow, and Klein. As their formalization is central to ours, we begin by discussing their work and highlight the differences involved in our verification.

Nipkow presents in [17] a framework for the formal verification of bytecode verification based on dataflow algorithms. This framework formalizes an operational semantics of the JVM and introduces the notion of a well-typing for bytecode programs. Based on the work of Pusch [23], he presents a proof that every program possessing a well-typing is type safe, i.e., the JVM will not reach a state where instructions operate on ill-typed data during the program's execution. In particular, Nipkow verifies an instance of a bytecode verifier that is based on Kildall's algorithm and formalizes the approach taken by Sun's bytecode verifier. He shows that the program types computed by the algorithm, which do not contain the Err element, are well-typings.

The work of Pusch and Nipkow focuses on the main concepts of the JVM; the formalization of the type system is restricted to integers and booleans as primitive types and references to classes that are not interfaces or array types. They also omit subroutines, exception handling, and object initialization from their formalization. Despite these omissions, their theory is substantial and encompasses more than 8000 lines of definition and proof.

In our work, we build directly on this formalization. This provides a good basis for validating the concept of static analysis by model checking, albeit under the same omissions. Although there is no conceptual difficulty in adding the missing features, doing so would be a large undertaking.[2] Note that Nipkow's notation, carried over in our formalization, differs slightly from the notation of [15] that we used in the first part of this paper. These differences are minor (e.g. INT versus integer) and self-explanatory.

The relationship between our development and Nipkow's can be seen under three different aspects: instantiation, modification, and extension. First, we instantiate Nipkow's abstract framework in that we reuse his foundational theories on semilattices, most of his formalization of the JVM and its type system, and his results on type safety. This allows us to substantially simplify our task: we can reduce the correctness of our approach (successfully model-checked code is type safe) to showing that when a program passes our bytecode verifier, then it possesses a well-typing. In doing so, we instantiate Nipkow's framework with a modified formalization of the notion of a state type (cf. Section 6.1). This particular instantiation is motivated by the fact that model checking is less restrictive than dataflow analysis and allows multiple state types per program point, depending on the control-flow paths leading to the program point (cf. Section 2.1).

[2] Recently these omissions have been lifted and the formalization comprises almost all the features of the Java Virtual Machine (see [12, 13]).

Second, to build the above instance, we modify the notion of the stack type to allow stack elements of incompatible types, thus generalizing the notion of the JVM state type. Of course, making such modifications requires reestablishing some of Nipkow's results. For example, given our change to the JVM state type, we needed to reprove Nipkow's type-safety theorem; however, we could reuse almost all of his proofs, with only minor modifications.

Finally, to support model checking, new extensions are required. For example, we must develop a theory that formalizes the syntax and semantics of linear temporal logic (LTL). This allows us to characterize abstractly, i.e. independent of the model-checking algorithm used, the properties that are checked.

The above describes the changes and extensions we made to Nipkow's theories to formalize our approach. There is one point, however, where our formalization deviates from our implemented system. Namely, to allow the computation of the supremum of two stacks at a program point, our formalization requires the stack size to be constant. As a result, our current formalization constitutes a midpoint, in terms of the set of programs accepted, between Nipkow's formalization and our implementation. By further generalizing the state type, it should be possible to lift this restriction. Alternatively, we could eliminate this difference between our system and our formalization by simply having the model checkers enforce this restriction. The downside of this is that the model checker would then reject some additional programs. But these are also rejected by bytecode verifiers based on dataflow analysis, like Sun's.

We can summarize our development, which is subdivided into six areas, as shown in Table I. Based on (1) *preliminary definitions* and (2) *semilattice-theoretic foundations*, we define (3) the *JVM model*, which includes the JVM type system, the JVM abstract semantics, and the definition of the JVM state type. In (4), the *model-checking framework*, we define Kripke structures and traces, which we later use to formalize model checking. Afterwards, we define the translation of bytecode programs into (5) *finite transition systems*. We use the abstract semantics of JVM programs to define the transition relations and the JVM type system to build LTL formulae for model checking. Finally, we state and prove in (6) our *main theorem*.

5. Foundational background

We now present background concepts necessary for our formalization. The subsections 5.1, 5.2, and 5.4–5.6 describe (unmodified) parts of Nipkow's formalization and are summarized here for the sake of completeness.

Table I. Overview of the theories constituting our formalization

Theories	**Status**
JBasis (1), Type (1), Decl (1), TypeRel (1), State (1),WellType (1), Conform (1), Value(1), Semilat (2), Err (2), Opt (2), Product(2), JType (3), BVSpec (3)	unchanged
Listn (2)	the stack model is changed
JVMInstructions (1), JVMExecInstr (1), JVMExec (1), JVMType (3), Step (3),	modified due to the changes in the stack model
Semilat2 (2), Kripke (4), LTL (4), ModelChecker (5), JVM_MC (6)	new

5.1. Basic types

We employ basic types and definitions of Isabelle/HOL. Types include *bool*, *nat*, *int*, the polymorphic types α *set* and α *list*, and the product type $\alpha \times \beta$. Note that $\times$ is used both on the type and on the set level. We employ a number of standard functions on (cons) lists including a conversion function set from lists to sets, infix operators # and @ to build and concatenate lists, and a function size to denote the length of a list. $xs!i$ denotes the i-th element of a list xs and $xs[i := x]$ overwrites the i-th element of xs with x. Finally, we use records to build tuples and functional images over sets: $(|\ \mathsf{a} :: \alpha,\ \mathsf{b} :: \beta\ |)$ denotes the record type containing the components a and b of types α and β respectively and $\mathsf{f}\ `\ A$ denotes the image of the function f over the set A.

5.2. Partial orders and semilattices

A partial order is a binary predicate of type $\alpha\ ord = \alpha \to \alpha \to bool$. We write $x \leq_r y$ for $r\ x\ y$ and $x <_r y$ for $x \leq_r y \wedge x \neq y$. We say that $r :: \alpha\ ord$ is a **partial order** iff r is reflexive, antisymmetric, and transitive. We formalize this using the predicate order $:: \alpha\ ord \to bool$. The top element of a set T with respect to the partial order r is characterized by the predicate top $:: \alpha\ ord \to \alpha \to bool$, which is defined by $\mathsf{top}\ r\ T \equiv \forall x.\, x \leq_r T$. Given the types $\alpha\ binop = \alpha \to \alpha \to \alpha$ and $\alpha\ sl = \alpha\ set \times \alpha\ ord \times \alpha\ binop$ and the supremum notation $x +_f y = f\ x\ y$, we say that $(A, r, f) :: \alpha\ sl$ is a (**supremum**) **semilattice** iff the predicate

$\mathsf{semilat} :: \alpha\ sl \to bool$ holds, where

$$\begin{array}{l}\mathsf{semilat}(A,\ r,\ f) \equiv \\ \quad \mathsf{order}\ r\ \wedge\ \mathsf{closed}\ A\ f\ \wedge \\ \quad (\forall\, x\, y \in A.\ x \leq_r x +_f y)\ \wedge\ (\forall\, x\, y \in A.\ y \leq_r x +_f y)\ \wedge \\ \quad (\forall\, x\, y\, z \in A.\ x \leq_r z\ \wedge\ y \leq_r z \longrightarrow x +_f y \leq_r z)\end{array}$$

and $\mathsf{closed}\ A\ f\ \equiv\ \forall\, x\, y \in A.\ x +_f y \in A$.

5.3. Least upper bounds of sets

The above definitions are for reasoning about the supremum of two elements using a binary operator. We define an additional theory Semilat2 to reason about the supremum of sets.

To build suprema over sets, we define the function $\mathsf{lift\_sup} :: \alpha\ sl \to \alpha \to \alpha \to \alpha \to \alpha$, written $\mathsf{lift\_sup}\ (A, r, f)\ T\ x\ y\ =\ x\ \uplus_{(A,r,f),T}\ y$, that lifts the binary operator $f :: \alpha\ binop$ over the type α.

$$\begin{array}{l}x\ \uplus_{(A,r,f),T}\ y\ \equiv\ \mathtt{if}\ (\mathsf{semilat}\,(A,r,f)\ \wedge\ \mathsf{top}\ r\ T)\ \mathtt{then} \\ \qquad\qquad\qquad \mathtt{if}\ (x \in A\ \wedge\ y \in A)\ \mathtt{then}\ (x\ +_f\ y)\ \mathtt{else}\ T \\ \qquad\qquad \mathtt{else}\ \mathsf{arbitrary}\end{array}$$

To reason about the least upper bounds of sets, we introduce the bottom element $B :: \alpha$, defined by $\mathsf{bottom}\ r\ B\ \equiv\ \forall\, x.\, B \leq_r x$. We use the Isabelle/HOL function $\mathsf{fold} :: (\beta \to \alpha \to \alpha) \to \alpha \to \beta\ set \to \alpha$ to build the least upper bounds of sets and we write $\bigsqcup_{(A,r,f),T,B}\ A'$ for $\mathsf{fold}\ (\lambda\, x\, y.\ x\ \uplus_{(A,r,f),T}\ y)\ B\ A'$. We observe that $\bigsqcup_{(A,r,f),T,B}\ A'$ is a least upper bound of A'.

The following lemma states (1) the monotonicity of the supremum over sets and extends (2) the semi-homomorphism property $g\, x +_f y\ \leq_r\ g\,(x) +_f g\,(y)$ to sets. (3) combines (1) and (2) and will be used later in our correctness proof.

LEMMA 1. *Let (A, r, f) be a finite semilattice with top element $T \in A$, bottom element $B \in A$, and subsets $A' \subseteq A$ and $A'' \subseteq A$. Furthermore let $g :: \alpha \to \alpha$, where α is the element type of the semilattice, be a strict function that is closed on A and that satisfies the semi-homomorphism property, i.e., $g\ B = B$, $\forall\, x \in A.\ g\ x \in A$, and $\forall\, x\, y \in A.\ g(x +_f y)\ \leq_r\ g(x) +_f g(y)$. Then it holds that:*

$$A'' \subseteq A' \longrightarrow \left(\bigsqcup_{(A,r,f),T,B}\ A''\right)\ \leq_r\ \left(\bigsqcup_{(A,r,f),T,B}\ A'\right) \qquad (1)$$

$$g\left(\bigsqcup_{(A,r,f),T,B}\ A'\right)\ \leq_r\ \bigsqcup_{(A,r,f),T,B}\ (g\ `\ A') \qquad (2)$$

$$(g\ `\ A'') \subseteq A' \longrightarrow g\left(\bigsqcup_{(A,r,f),T,B}\ A''\right)\ \leq_r\ \left(\bigsqcup_{(A,r,f),T,B}\ A'\right). \qquad (3)$$

5.4. The error type and *err*-semilattices

The theory Err defines an error element, which we will use to model the situation where the supremum of two elements does not exist. We introduce both a datatype and a corresponding construction on sets.

$$\begin{array}{l}\textbf{datatype}\ \alpha\ err \equiv \mathsf{Err} \mid \mathsf{OK}\ \alpha\\ \mathsf{err}\ A \equiv \{\mathsf{Err}\} \cup \{\mathsf{OK}\, a \mid a \in A\}\end{array}$$

Orderings r on α can be lifted to $\alpha\ err$ by making Err the top element.

$$\begin{array}{lllll}\mathsf{le}\ r\ \_ & \mathsf{Err} & = & \mathsf{True}\\ \mathsf{le}\ r\ \mathsf{Err} & (\mathsf{OK}\ y) & = & \mathsf{False}\\ \mathsf{le}\ r\ (\mathsf{OK}\ x) & (\mathsf{OK}\ y) & = & x \leq_r y\end{array}$$

We now employ the following lifting function

$$\begin{array}{lllll}\mathsf{lift2} :: (\alpha \to \beta \to \gamma\ err) \to \alpha\ err \to \beta\ err \to \gamma\ err\\ \mathsf{lift2}\ f\ (\mathsf{OK}\ x)\ (\mathsf{OK}\ y) & = & f\ x\ y\\ \mathsf{lift2}\ f\ \_\ \_ & = & \mathsf{Err}\end{array}$$

and a function that lifts supremum functions f on α to supremum functions on $\alpha\ err$

$$\begin{array}{l}\mathsf{sup} :: (\alpha \to \beta \to \gamma) \to (\alpha\ err \to \beta\ err \to \gamma\ err)\\ \mathsf{sup}\ f \equiv \mathsf{lift2}\ (\lambda x\, y.\ \mathsf{OK}\, (x\ +_f\ y))\end{array}$$

to define a new notion of an *err*-semilattice, which is a variation of a semilattice with a top element. It suffices to say how the ordering and the supremum are defined over non-top elements and hence we represent a semilattice with top element Err as a triple of type *esl*: $\alpha\ esl = \alpha\ set \times \alpha\ ord \times \alpha\ ebinop$, where $\alpha\ ebinop = \alpha \to \alpha \to \alpha\ err$. We also define conversion functions between the types *sl* and *esl*.

$$\begin{array}{l}\mathsf{esl} :: \alpha\ sl \to \alpha\ esl\\ \mathsf{esl}\, (A, r, f) = (A,\ r,\ \lambda x\, y.\ \mathsf{OK}\, (f\ x\ y))\end{array}$$

$$\begin{array}{l}\mathsf{sl} :: \alpha\ esl \to \alpha\ err\ sl\\ \mathsf{sl}\, (A, r, f) = (\mathsf{err}\ A,\ \mathsf{le}\ r,\ \mathsf{lift2}\ f)\end{array}$$

Finally we define $L :: \alpha\ esl$ to be an *err*-**semilattice** iff sl L is a semilattice. It follows that esl L is an *err*-semilattice if L is a semilattice.

5.5. The option type

Theory Opt introduces the type *option* and the set opt as duals to the type *err* and the set err.

$$\begin{array}{l}\textbf{datatype}\ \alpha\ option \equiv \mathsf{None} \mid \mathsf{Some}\ \alpha\\ \mathsf{opt}\ A \equiv \{\mathsf{None}\} \cup \{\mathsf{Some}\ a \mid a \in A\}\end{array}$$

The theory also defines a function that lifts orderings r on α to orderings on α *option* by making None the bottom element.

$$\begin{array}{llll}\mathsf{le}\ r\ \mathsf{None} & \_ & = & \mathsf{True}\\ \mathsf{le}\ r\ (\mathsf{Some}\ x) & \mathsf{None} & = & \mathsf{False}\\ \mathsf{le}\ r\ (\mathsf{Some}\ x) & (\mathsf{Some}\ y) & = & x \leq_r y\end{array}$$

Additionally it defines a function that lifts binary operators f of type α *ebinop* to binary operators of type α *option ebinop*.

$$\begin{array}{llll}\mathsf{sup}\ f\ \mathsf{None} & z & = & \mathsf{OK}\ z\\ \mathsf{sup}\ f\ z & \mathsf{None} & = & \mathsf{OK}\ z\\ \mathsf{sup}\ f\ (\mathsf{Some}\ x) & (\mathsf{Some}\ y) & = & \mathsf{case}\ (f\ x\ y)\ \mathsf{of}\\ & & & \quad \mathsf{Err} \Rightarrow \mathsf{Err}\\ & & & \mid \mathsf{OK}\ s \Rightarrow \mathsf{OK}(\mathsf{Some}\ s)\end{array}$$

Note that the function $\mathsf{esl}\,(A, r, f) = (\mathsf{opt}\ A, \mathsf{le}\ r, \mathsf{sup}\ f)$ maps *err*-semilattices to *err*-semilattices.

5.6. Products

Theory Product provides what is known as *coalesced* products, where the top elements of both components are identified.

$\mathsf{esl} :: \alpha\ esl \to \beta\ esl \to (\alpha \times \beta)\ esl$
$\mathsf{esl}\ (A, r_A, f_A)\ (B, r_B, f_B) = (A \times B, \mathsf{le}\ r_A\ r_B, \mathsf{sup}\ f_A\ f_B)$
$\mathsf{sup} :: \alpha\ ebinop \to \beta\ ebinop \to (\alpha \times \beta)\ ebinop$
$\mathsf{sup}\ f\ g = \lambda(a_1, b_1)\,(a_2, b_2).\ \mathsf{Err.sup}\ (\lambda x\, y.\,(x, y))\ (a_1 +_f a_2)\ (b_1 +_g b_2)$

The ordering function $\mathsf{le} :: \alpha\ ord \to \beta\ ord \to (\alpha \times \beta)\ ord$ is defined as expected. If both L_1 and L_2 are *err*-semilattices, then so is $\mathsf{esl}\,L_1\,L_2$.

5.7. Lists of fixed length

For our application, we model the JVM stack differently from Nipkow. In particular, to support polyvariant analysis, we must associate multiple stack types to each program point within the program. This modification takes place in the theory Listn in which lists of fixed length over a given

set are defined. In Isabelle/HOL this is formalized as a set rather than a type, namely

$$\mathsf{list}\ n\ A\ \equiv\ \{xs \mid \mathsf{size}\ xs = n\ \wedge\ \mathsf{set}\ xs \subseteq A\}\,.$$

This set can be turned into a semilattice in a componentwise manner, essentially by viewing it as an n-fold Cartesian product.

$$\begin{array}{l}\mathsf{sl} :: nat\ \to\ \alpha\ sl\ \to\ \alpha\ list\ sl\\ \mathsf{sl}\ n\ (A, r, f) = (\mathsf{list}\ n\ A,\ \mathsf{le}\ r,\ \mathsf{map2}\ f)\\ \\ \mathsf{le} :: \alpha\ ord\ \to\ \alpha\ list\ ord\\ \mathsf{le}\ r\ =\ \mathsf{list\_all2}\ (\lambda\, x\, y.\, x \leq_r y)\end{array}$$

Here the auxiliary functions $\mathsf{map2} :: (\alpha \to \beta \to \gamma) \to \alpha\ list \to \beta\ list \to \gamma\ list$ and $\mathsf{list\_all2} :: (\alpha \to \beta \to bool) \to \alpha\ list \to \beta\ list \to bool$ pointwise extend binary functions and predicates on elements to binary functions and binary predicates on lists. We write $xs \leq_{[r]} ys$ for $xs \leq_{(\mathsf{le}\ r)} ys$ and $xs +_{[f]} ys$ for $xs +_{(\mathsf{map2}\ f)} ys$. If L is a semilattice, then so is $\mathsf{sl}\ n\ L$.

To combine lists of different lengths, we define the function

$$\begin{array}{l}\mathsf{sup} :: (\alpha \to \beta \to \gamma\ err) \to \alpha\ list \to \beta\ list \to \gamma\ list\ err\\ \mathsf{sup}\ f\ xs\ ys\ =\ \mathtt{if}\ (\mathsf{size}\ xs\ =\ \mathsf{size}\ ys)\\ \qquad\qquad\qquad\ \mathtt{then}\ \mathsf{OK}\,(\mathsf{map2}\ f\ xs\ ys)\\ \qquad\qquad\qquad\ \mathtt{else}\ \mathsf{Err}.\end{array}$$

Note that in our JVM formalization, the supremum of two lists xs and ys of equal length returns a result of the form $\mathsf{OK}\ zs$ with $zs!i\ =\ \mathsf{Err}$, when the supremum of two corresponding elements $xs!i$ and $ys!i$ equals Err. This differs from sup in Nipkow's formalization, which returns Err in this case. Below we present the function upto_esl that maps a natural number n and a semi-lattice L to an err-semilattice of all lists over L of length up to n.

$$\begin{array}{l}\mathsf{upto\_esl} :: nat\ \to \alpha\ sl \to \alpha\ list\ esl\\ \mathsf{upto\_esl}\ n\ =\ \lambda\,(A, r, f).\ (\cup_{i \leq n}\, \mathsf{list}\ i\ A,\ \mathsf{le}\ r,\ \mathsf{sup}\ f)\end{array}$$

6. The JVM model

We now show how the JVM can be formalized for the purpose of polyvariant dataflow analysis. In Section 6.1, our formalization adopts unchanged Nipkow's formalization of the JVM and the JVM type system. Using this, we define our modified program state type and construct a semilattice whose carrier set consists of elements of this type. Based on this modified

program state type, we redefine in Section 6.2 the syntax and abstract semantics of JVM programs and consequently also redefine the JVM abstract execution function and the notion of a well-typed method in Section 6.3.

6.1. Type system and well-formedness

The theory Types defines the types of our JVM. Our machine supports operations over elements of the type *ty*: the void type, integers, booleans, null references, and class types (based on the type *cname* of class names).

$$\textbf{datatype}\ ty\ \equiv\ \mathsf{Void}\ |\ \mathsf{Integer}\ |\ \mathsf{Boolean}\ |\ \mathsf{NullT}\ |\ \mathsf{Class}\ cname$$

The theory Decl defines class declarations and programs. Based on the type *mname* of method names and *vname* of variable names, we model a JVM program $\mathsf{P} :: \gamma\ prog$ as a list of class files, where

$$\begin{array}{lcl}\gamma\ prog & = & \gamma\ cdecl\ list\\ \gamma\ cdecl & = & cname \times \gamma\ class\\ \gamma\ class & = & cname \times (vname \times ty)\ list \times \gamma\ mdecl\ list\\ \gamma\ mdecl & = & (mname \times ty\ list) \times ty \times \gamma.\end{array}$$

Each class file records its class name, the name of its super class, a list of field declarations, and a list of method declarations. The type *cname* is assumed to have a distinguished element Object.

Our program formalization gives rise to a subclass relation subcls1 and subclasses induce a subtype relation $\mathsf{subtype} :: \gamma\ prog \to ty \to ty \to bool$. Based on the subtype P relation, we define the supremum on types $\mathsf{sup} :: ty \to ty \to ty\ err$ as is standard. As abbreviations, we define $\mathsf{types}\ P\ =\ \{\tau\ |\ \mathsf{is\_type}\ P\ \tau\}$ and $\tau_1 \sqsubseteq_P \tau_2\ =\ \tau_1 \leq_{\mathsf{subtype}\ P} \tau_2$. Below, we use the predicate $\mathsf{is\_class}\ P\ C$ to express that the class C is in the program P.

Well-formedness of JVM programs is defined by context conditions that can be checked statically prior to bytecode verification. We formalize this using a predicate $\mathsf{wf\_prog} :: \gamma\ wf\_mb \to \gamma\ prog \to bool$, where $\gamma\ wf\_mb\ =\ \gamma\ prog \to cname \to \gamma\ mdecl \to bool$. Informally, $\mathsf{wf\_prog}\ wf\_mb\ P$ means that all methods of the program are well-formed with respect to the predicate *wf_mb* (given as the first parameter), subcls1 P is univalent (i.e. subcls1 P represents a single inheritance hierarchy) and acyclic, and both $(\mathsf{subcls1}\ P)^{-1}$ and $(\mathsf{subtype}\ P)^{-1}$ are well-founded. The definition of wf_prog is given in [17].

The following lemma holds for all programs P and predicates *wf_mb*.

LEMMA 2.

$$\mathsf{wf\_prog}\ wf\_mb\ P\ \longrightarrow\ \mathsf{semilat}\,(\mathsf{sl}\,(\mathsf{types}\ P,\ \mathsf{subtype}\ P,\ \mathsf{sup}\ P))\ \wedge\ \mathsf{finite}\,(\mathsf{types}\ P)$$

We will use the semilattice $\mathsf{sl}\,(\mathsf{types}\ P, \mathsf{subtype}\ P, \mathsf{sup}\ P)$ to construct a semilattice with the carrier set of program states.

The JVM is a stack machine where each activation record consists of a stack for expression evaluation and a list of local variables (called **registers**). The abstract semantics, which operates with types as opposed to values, records the type of each stack element and each register. At different program points, a register may hold incompatible types, e.g. an integer or a reference, depending on the computation path that leads to that point. This facilitates the reuse of registers and is modeled by the HOL type *ty err*, where $\mathsf{OK}\ \tau$ represents the type τ and Err represents the inconsistent type. In our JVM formalization, the elements of the stack can also be reused. Thus a state type of our abstract JVM is a pair of lists,

$$\mathit{state\_type} \;=\; \mathit{ty\ err\ list} \;\times\; \mathit{ty\ err\ list}\,,$$

which model an expression stack and a list of registers. Note that this type differs from the type *ty list* $\times$ *ty err list* presented in [17], where the stack only holds the values that can actually be used.

We now define the type of the program state as *state_type option err*, where OK None indicates an unreachable program point (dead code), OK (Some s) is a normal state type s, and Err is an error. In the following, we use the type association *state* $=$ *state_type option err*. Turning *state* into a semilattice structure is easy because all of its constituent types are (*err*-)semilattices. The set of operand stacks forms the carrier set of an *err*-semilattice because the supremum of stacks of different size is Err; the set of lists of registers forms the carrier set of a semilattice because the number of registers is fixed.

$$\begin{array}{lcl}
\mathsf{stk_esl} & :: & \gamma\ \mathit{prog} \to \mathit{nat} \to \mathit{ty\ err\ list\ esl} \\
\mathsf{stk_esl}\ P\ \mathit{maxs} & \equiv & \mathsf{upto_esl}\ \mathit{maxs}\ (\mathsf{sl}\,(\mathsf{types}\ P, \mathsf{subtype}\ P, \mathsf{sup}\ P)) \\
\\
\mathsf{reg_sl} & :: & \gamma\ \mathit{prog} \to \mathit{nat} \to \mathit{ty\ err\ list\ sl} \\
\mathsf{reg_sl}\ P\ \mathit{maxr} & \equiv & \mathsf{Listn.sl}\ \mathit{maxr}\ (\mathsf{sl}\ (\mathsf{types}\ P, \mathsf{subtype}\ P, \mathsf{sup}\ P))
\end{array}$$

Since any error on the stack must be propagated, the stack and registers are combined in a coalesced product using Product.esl and then are embedded into *option* and *err* to form a semilattice. This is accomplished by the function sl.

$$\begin{array}{lcl}
\mathsf{sl} & :: & \gamma\ \mathit{prog} \to \mathit{nat} \to \mathit{nat} \to \mathit{state\ sl} \\
\mathsf{sl}\ P\ \mathit{maxs}\ \mathit{maxr} & \equiv & \mathsf{Err.sl}\ (\mathsf{Opt.esl}\ (\mathsf{Product.esl} \\
 & & \qquad (\mathsf{stk_esl}\ P\ \mathit{maxs}) \\
 & & \qquad (\mathsf{Err.esl}\ \mathsf{reg_sl}\ P\ \mathit{maxr})))
\end{array}$$

In the following we abbreviate $\mathsf{sl}\ P\ \mathit{maxs}\ \mathit{maxr}$ as sl, $\mathsf{states}\ P\ \mathit{maxs}\ \mathit{maxr}$ as states, $\mathsf{le}\ P\ \mathit{maxs}\ \mathit{maxr}$ as le, and $\mathsf{sup}\ P\ \mathit{maxs}\ \mathit{maxr}$ as sup. Using properties about semilattices, it is easy to prove the following lemma:

LEMMA 3.

$\forall\, P\ maxs\ maxr.$ sl = (states, le, sup),
$\forall\, wf\_mb\ P\ maxs\ maxr.$ (wf_prog $wf\_mb\ P$) $\longrightarrow$ semilat (states, le, sup),
$\forall\, wf\_mb\ P\ maxs\ maxr.$ (wf_prog $wf\_mb\ P$) $\longrightarrow$ finite (states),
$\forall\, P\ maxs\ maxr.$ Err $\in$ states,
$\forall\, P\ maxs\ maxr.$ top le Err,
$\forall\, P\ maxs\ maxr.$ (OK None) $\in$ states, *and*
$\forall\, P\ maxs\ maxr.$ bottom le (OK None).

6.2. Program syntax and abstract semantics

The theory JVMInstructions defines the JVM instruction set. In our JVM formalization, the polymorphic instructions Load, Store, and CmpEq are replaced with instructions that have their counterparts in the Sun JVM instruction set, i.e. one such instruction for each base type (see the explanation at the end of Section 6.3).

datatype *instr* =	iLoad *nat*	bLoad *nat*	aLoad *nat*
Invoke *cname mname*	iStore *nat*	bStore *nat*	aStore *nat*
Getfield *vname cname*	iIfcmpeq *nat*	LitPush *val*	Dup
Putfield *vname cname*	bIfcmpeq *nat*	New *cname*	Dup_x1
Checkcast *cname*	aIfcmpeq *nat*	Return	Dup_x2
Pop	Swap	IAdd	Goto *int*

We instantiate the polymorphic type of programs γ *prog* using the type $nat \times nat \times instr\ list$, which reflects that bytecode methods contain information about the class file attributes `max_stack` and `max_locals`, and the instructions of the method body. We model the type-level execution of a single instruction with step′ :: $instr \times jvm\_prog \times state\_type \to state\_type$, where $jvm\_prog = (nat \times nat \times instr\ list)\ prog$. Below we show the definition of step′ for selected instructions. For instance, the abstract Getfield instruction puts the type of the referenced field on the operand stack (the type is the second component of the field declaration obtained by the(field$(P, C)\ fn$)).

$$
\begin{aligned}
&\text{step}'\ (\text{iLoad}\ n,\ P,\ (st, reg)) &&= ((reg!n)\#st, reg)\\
&\text{step}'\ (\text{iStore}\ n,\ P,\ (\tau_1\#st_1, reg)) &&= (st_1, reg[n := \tau_1])\\
&\text{step}'\ (\text{iAdd},\ P,\ ((\text{OK Integer})\#(\text{OK Integer})\#st_1, reg)) &&=\\
&&&\quad ((\text{OK Integer})\#st_1, reg)\\
&\text{step}'\ (\text{Getfield}\ fn\ C,\ P,\ (\tau_o\#st_1, reg)) &&=\\
&&&\quad (\text{OK}(\text{snd}(\text{the}(\text{field}(P, C)\ fn)))\#st_1, reg)\\
&\text{step}'\ (\text{iIfcmpeq}\ b,\ P,\ (\tau_1\#\tau_2\#st_1, reg)) &&= (st_1, reg)\\
&\text{step}'\ (\text{Return},\ P,\ (st, reg)) &&= (st, reg)
\end{aligned}
$$

Note that the execution of the Return instruction is modeled by a self-loop. This will be useful when we model traces as *infinite* sequences of program states. Finite sequences can occur only in ill-formed programs, where an instruction has no successors.

6.3. Bytecode verifier specification

We now define a predicate $\mathsf{app}' :: instr \times jvm\_prog \times nat \times ty \times state\_type \to bool$, which expresses the applicability of the function step' to a state type (st, reg); again we show only a few cases.

$$
\begin{array}{lll}
\mathsf{app}'\,(i,\ P,\ maxs,\ rT,\ (st,\ reg)) \equiv \mathtt{case}\ i\ \mathtt{of} & & \\
\quad \mathsf{iLoad}\ n & \Rightarrow & n < \mathsf{size}\ reg\ \wedge\ reg!n = (\mathsf{OK}\ \mathsf{Integer}) \wedge \\
 & & \quad \mathsf{size}\ st < maxs \\
\mid \mathsf{iStore}\ n & \Rightarrow & \exists\, \tau\ st_1.\ n < \mathsf{size}\ reg\ \wedge\ st = \tau \# st_1 \wedge \\
 & & \quad\ \tau = (\mathsf{OK}\ \mathsf{Integer}) \\
\mid \mathsf{iAdd} & \Rightarrow & \exists\, st_1.\ st = (\mathsf{OK}\ \mathsf{Integer}) \# (\mathsf{OK}\ \mathsf{Integer}) \# st_1 \\
\mid \mathsf{Getfield}\ fn\ C & \Rightarrow & \exists\, \tau_o\ \tau_f\ st_2.\ st = (\mathsf{OK}\ \tau_o) \# st_2\ \wedge\ \mathsf{is\_class}\ P\ C \wedge \\
 & & \quad\ \mathsf{field}\ (P, C)\ fn = \mathsf{Some}\,(C,\ \tau_f)\ \wedge\ \tau_o \sqsubseteq_P \mathsf{Class}\ C \\
\mid \mathsf{iIfcmpeq}\ b & \Rightarrow & \exists\, st_1.\ st = (\mathsf{OK}\ \mathsf{Integer}) \# (\mathsf{OK}\ \mathsf{Integer}) \# st_1 \\
\mid \mathsf{Return} & \Rightarrow & \exists\, \tau\ st_1.\ st = \tau \# st_1\ \wedge\ \tau \sqsubseteq_P rT
\end{array}
$$

Furthermore, we introduce a successor function $\mathsf{succs} :: instr \to nat \to nat\ list$, which computes the possible successor instructions of a program point with respect to a given instruction.

$$
\begin{array}{lll}
\mathsf{succs}\ i\ p \equiv & \mathtt{case}\ i\ \mathtt{of} & \\
 & \quad \mathsf{Return} & \Rightarrow [p] \\
 & \mid \mathsf{Goto}\ b & \Rightarrow [p + b] \\
 & \mid \mathsf{aIfcmpeq}\ b & \Rightarrow [p + 1,\ p + b] \\
 & \mid \mathsf{bIfcmpeq}\ b & \Rightarrow [p + 1,\ p + b] \\
 & \mid \mathsf{iIfcmpeq}\ b & \Rightarrow [p + 1,\ p + b] \\
 & \mid \_ & \Rightarrow [p + 1]
\end{array}
$$

We use succs to construct the transition function of a finite transition system. To reason about the boundedness of succs, we define the predicate $\mathsf{bounded} :: (nat \to nat\ list) \to nat \to bool$, where $\mathsf{bounded}\ f\ n \equiv \forall p < n.\ \forall q \in \mathsf{set}\,(f\ p).\ q < n$.

Let $method\_type = state\_type\ option\ list$. We now define a well-typedness condition for bytecode methods. The predicate wt_method formalizes that given a program P, class C, method parameter list pTs, method return type rT, maximal stack size $maxs$, and the number of uninitialized registers $maxl$, a bytecode instruction sequence ins is well-

typed with respect to a given method type ϕ.

$$
\begin{array}{l}
\mathsf{wt\_method} :: \mathit{jvm\_prog} \to \mathit{cname} \to \mathit{ty\ list} \to \mathit{ty} \to \mathit{nat} \to \mathit{nat} \to \\
\qquad \mathit{instr\ list} \to \mathit{method\_type} \to \mathit{bool} \\
\mathsf{wt\_method}\ P\ C\ pTs\ rT\ maxs\ maxl\ ins\ \phi\ \equiv \\
0 < \mathsf{size}\ ins \wedge \\
(\mathsf{Some}\ ([], (\mathsf{OK}\ (\mathsf{Class}\ C))\#(\mathsf{map}\ \mathsf{OK}\ pTs)@(\mathsf{replicate}\ maxl\ \mathsf{Err}))) \\
\qquad\qquad \leq_{\mathsf{le\_state\_opt}}\ \phi!0 \wedge \\
(\forall\, pc.\, pc < \mathsf{size}\ ins\ \longrightarrow \\
\quad (\mathsf{app}\ (ins!pc)\ P\ maxs\ rT\ (\phi!pc)) \wedge \\
\quad (\forall\, pc' \in \mathsf{set}\,(\mathsf{succs}\ (ins!pc)\ pc).\, (pc' < \mathsf{size}\ ins) \wedge \\
\qquad\qquad (\mathsf{step}\ (ins!pc)\ P\ (\phi!pc)\ \leq_{\mathsf{le\_state\_opt}}\ \phi!pc'))
\end{array}
$$

Here, step and app are liftings of their primed counterparts from *state_type* to *state_type option*. Furthermore $\leq_{\mathsf{le\_state\_opt}}$ lifts the order on *state_type* to *state_type option*. Note that this definition of a well-typed method is in the style of Pusch [23].

The last element of our JVM model is the definition of the abstract execution function exec.

$$
\begin{array}{lcl}
\mathsf{exec} :: \mathit{jvm\_prog} \to \mathit{nat} \to \mathit{ty} \to \mathit{instr\ list} \to \mathit{nat} \to \mathit{state} \to \mathit{state} & & \\
\mathsf{exec}\ P\ maxs\ rT\ ins\ pc & \equiv & \lambda\, s.\, \mathtt{case}\ s\ \mathtt{of} \\
 & & \quad \mathsf{Err} \quad \Rightarrow\ \mathsf{Err} \\
 & & |\ \ \mathsf{OK}\ s' \ \Rightarrow\ \mathtt{if}\ \mathsf{app}\ (ins!pc)\ P\ maxs\ rT\ s' \\
 & & \qquad\qquad\ \ \mathtt{then}\ \mathsf{OK}\,(\mathsf{step}\ (ins!pc)\ P\ s') \\
 & & \qquad\qquad\ \ \mathtt{else}\ \mathsf{Err}
\end{array}
$$

Abbreviating exec P *maxs* rT *ins* pc as exec, we now have

LEMMA 4.

$$
\begin{array}{l}
\forall\, \mathit{wf\_mb}\ P\ maxs\ maxr\ ins\ pc.\, \forall\, s_1\ s_2 \in \mathsf{states}. \\
\quad (\mathsf{wf\_prog}\ \mathit{wf\_mb}\ P\ \wedge\ \mathsf{semilat\ sl}) \\
\qquad\qquad \longrightarrow\ \mathsf{exec}\ (s_1\ +_{\mathsf{sup}}\ s_2)\ \leq_{\mathsf{le}}\ (\mathsf{exec}\ s_1)\ +_{\mathsf{sup}}\ (\mathsf{exec}\ s_2).
\end{array}
$$

This lemma states a "semi-homomorphism" property of the exec function with respect to the le relation. To prove it we must show for our formalization of the JVM that if an instruction at a given program point is well-typed with respect to two arbitrary state types $x, y \in A$, then it is also well-typed with respect to the supremum $x +_f y$ of the two state types. This would have been impossible to prove using Nipkow's formalization of the JVM with polymorphic instructions; hence we have replaced the polymorphic instructions in the JVM instruction set with collections of monomorphic ones.

7. Model checking and correctness

We now verify the correctness of our approach. We first show how the abstract transition system of a bytecode method M is formalized as a Kripke structure K and how the applicability conditions are formalized as an LTL formula ψ. Second we show what it means for ψ to globally hold for K, i.e. $K \models \Box\,\psi$. Namely, we constructively show that when model checking succeeds, then there exists a type ϕ that is a well-typing for the method M.

7.1. Kripke structures and method abstraction

7.1.1. *Kripke structures*

A Kripke structure K consists of a non-empty set of states, a set of initial states, and a transition relation. A trace of K is an infinite sequence of states such that the first state is an initial state and pairs of successive states are in the transition relation. A state is reachable in K if it is contained in a trace of K. We also define a suffix function on traces, which is needed to define the semantics of LTL-formulae. These definitions are standard and their formalization in Isabelle/HOL is straightforward.

$$\alpha\ \mathit{kripke} = (|\ \mathsf{states} :: \alpha\ \mathit{set}, \mathsf{init} :: \alpha\ \mathit{set}, \mathsf{next} :: (\alpha \times \alpha)\ \mathit{set}\ |)$$

$$\mathsf{is\_kripke} :: \alpha\ \mathit{kripke} \to \mathit{bool}$$
$$\mathsf{is\_kripke}\ K \equiv \mathsf{states}\ K \neq \emptyset \wedge \mathsf{init}\ K \subseteq \mathsf{states}\ K \wedge \mathsf{next}\ K \subseteq \mathsf{states}\ K \times \mathsf{states}\ K$$

$$\alpha\ \mathit{trace} = \mathit{nat} \to \alpha$$

$$\mathsf{is\_trace} :: \alpha\ \mathit{kripke} \to \alpha\ \mathit{trace} \to \mathit{bool}$$
$$\mathsf{is\_trace}\ K\ t \equiv t\ 0 \in \mathsf{init}\ K \wedge \forall i.\ (t\ i, t\ (\mathsf{Suc}\ i)) \in \mathsf{next}\ K$$

$$\mathsf{traces} :: \alpha\ \mathit{kripke} \to \alpha\ \mathit{trace}\ \mathit{set}$$
$$\mathsf{traces}\ K \equiv \{t \mid \mathsf{is\_trace}\ K\ t\}$$

$$\mathsf{reachable} :: \alpha\ \mathit{kripke} \to \alpha \to \mathit{bool}$$
$$\mathsf{reachable}\ K\ q \equiv \exists t\, i.\ \mathsf{is\_trace}\ K\ t \wedge q = t\ i$$

$$\mathsf{suffix} :: \alpha\ \mathit{trace} \to \mathit{nat} \to \alpha\ \mathit{trace}$$
$$\mathsf{suffix}\ t\ i \equiv \lambda j.\ t\ (i + j)$$

7.1.2. *Method abstraction*

Using the above definitions, we formalize the abstraction of bytecode methods as a finite Kripke structure over the type $\mathit{abs\_state} = \mathit{nat} \times \mathit{state\_type}$. We generate the Kripke structure using the function abs_method, which given a program P, class name C, method parameter list pTs, return type rT, the number of uninitialized registers $maxl$, and a bytecode

instruction sequence *ins*, yields a Kripke structure of type *abs_state kripke*.

$$\begin{aligned}&\mathsf{abs\_method} :: \mathit{jvm\_prog} \to \mathit{cname} \to \mathit{ty\ list} \to \mathit{ty} \to \mathit{nat} \to \\ &\qquad \mathit{instr\ list} \to \mathit{abs\_state\ kripke}\\ &\mathsf{abs\_method}\ P\ C\ \mathit{pTs}\ \mathit{rT}\ \mathit{maxl}\ \mathit{ins} \equiv \\ &\qquad (|\ \mathsf{states} = \mathsf{UNIV},\\ &\qquad\ \ \mathsf{init} = \mathsf{abs\_init}\ C\ \mathit{pTs}\ \mathit{maxl},\\ &\qquad\ \ \mathsf{next} = (\bigcup_{pc\in\{p.p<(\mathsf{size}\ \mathit{ins})\}} \mathsf{abs\_instr}\ (\mathit{ins}!\mathit{pc})\ P\ \mathit{pc})\ |)\end{aligned}$$

The set of states is modeled by the set UNIV of all elements of type *abs_state*. The set of initial states abs_init contains one element, which models the method entry, where the program counter is set to 0 and the stack is empty. At method entry, the list of local variables contains the this reference OK (Class C), the method's parameters map OK pTs, and $maxl$ uninitialized registers replicate $maxl$ Err.

$$\begin{aligned}&\mathsf{abs\_init} :: \mathit{cname} \to \mathit{ty\ list} \to \mathit{nat} \to \mathit{abs\_state\ set}\\ &\mathsf{abs\_init}\ C\ \mathit{pTs}\ \mathit{maxl} \equiv\\ &\quad \{\,(0,\ ([],\ (\mathsf{OK}\ (\mathsf{Class}\ C))\#(\mathsf{map\ OK}\ \mathit{pTs})@(\mathsf{replicate}\ \mathit{maxl}\ \mathsf{Err})))\,\}\end{aligned}$$

The relation next is generated by the function abs_instr, which uses step′ and succs, which together make up the transition relation. Note that the next relation is finite because both the type system and the number of storage locations (stack and local variables) of the abstracted method are finite.

$$\begin{aligned}&\mathsf{abs\_instr} :: \mathit{instr} \to \mathit{jvm\_prog} \to \mathit{nat} \to (\mathit{abs\_state} \times \mathit{abs\_state})\ \mathit{set}\\ &\mathsf{abs\_instr}\ i\ P\ \mathit{pc} \equiv \{((\mathit{pc}',\ q),\ (\mathit{pc}'',\ q'))\ |\ \ \mathit{pc}'' \in \mathsf{set}\,(\mathsf{succs}\ i\ \mathit{pc}') \wedge\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (q' = \mathsf{step}'\,(i,\ P,\ q)) \wedge\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{pc} = \mathit{pc}'\}\end{aligned}$$

7.2. Temporal logic and applicability conditions

7.2.1. *Temporal logic*

The syntax of LTL formulae is given by the datatype $\alpha\ \mathit{ltl}$.

$$\begin{aligned}\textbf{datatype}\ \alpha\ \mathit{ltl} \equiv\ &\mathsf{Tr} \mid \mathsf{Atom}\ (\alpha \to \mathit{bool}) \mid \mathsf{Neg}\ (\alpha\ \mathit{ltl})\\ &\mid \mathsf{Conj}\ (\alpha\ \mathit{ltl})\ (\alpha\ \mathit{ltl}) \mid \mathsf{Next}\ (\alpha\ \mathit{ltl})\\ &\mid \mathsf{Until}\ (\alpha\ \mathit{ltl})\ (\alpha\ \mathit{ltl})\end{aligned}$$

As is standard, the modalities eventually, $\Diamond :: \alpha\ \mathit{ltl} \to \alpha\ \mathit{ltl}$, and globally, $\Box :: \alpha\ \mathit{ltl} \to \alpha\ \mathit{ltl}$, can be defined as syntactic sugar.

$$\Diamond\,\varphi \equiv \mathsf{Until\ Tr}\ \varphi \qquad \Box\,\varphi \equiv \mathsf{Neg}\ (\Diamond\ (\mathsf{Neg}\ \varphi))$$

We now define the semantics of LTL using a satisfiability predicate

$$\_ \models \_ :: \alpha\ trace \to \alpha\ ltl \to bool\,,$$

where

$$\begin{array}{lcl}
t \models \mathsf{Tr} & = & \mathsf{True} \\
t \models \mathsf{Atom}\ p & = & p\,(t\ 0) \\
t \models \mathsf{Neg}\ \varphi & = & \neg\,(t \models \varphi) \\
t \models \mathsf{Conj}\ \varphi\ \psi & = & (t \models \varphi) \wedge (t \models \psi) \\
t \models \mathsf{Next}\ \varphi & = & \mathsf{suffix}\ t\ 1 \models \varphi \\
t \models \mathsf{Until}\ \varphi\ \psi & = & \exists j.\ (\mathsf{suffix}\ t\ j \models \psi) \wedge \\
 & & \quad (\forall i.\ i < j \longrightarrow \mathsf{suffix}\ t\ i \models \varphi)\,.
\end{array}$$

Furthermore, we say that a property φ is globally satisfied in the Kripke structure K if it is satisfied for all traces of K.

$$\begin{array}{l}
_ \models _ :: \alpha\ kripke \to \alpha\ ltl \to bool \\
K \models \varphi \equiv \forall t \in \mathsf{traces}\ K.\ t \models \varphi
\end{array}$$

7.2.2. *Applicability conditions*
The applicability conditions are expressed by an LTL formula of the form $\Box$ Atom φ.[3] We extract the formula φ from the bytecode, using the functions app_instr and app_method.

$$\begin{array}{l}
\mathsf{app_instr} :: instr \to jvm_prog \to nat \to nat \to nat \to ty \to \\
\qquad abs_state \to bool \\
\mathsf{app_instr}\ i\ P\ pc\ h\ maxs\ rT \equiv \lambda(p, q). \\
\quad p = pc \longrightarrow ((\mathsf{size}\ (\mathsf{fst}\ q)) = h) \wedge (\mathsf{app}'\ (i, P, maxs, rT, q))
\end{array}$$

$$\begin{array}{l}
\mathsf{app_method} :: jvm_prog \to nat\ list \to ty \to nat \to instr\ list \to \\
\qquad abs_state \to bool \\
\mathsf{app_method}\ P\ hs\ rT\ maxs\ ins \equiv \lambda(p, q). \\
\quad \forall pc < (\mathsf{size}\ ins).\ (\mathsf{app_instr}\ (ins!pc)\ P\ pc\ (hs!pc)\ maxs\ rT\ (p, q))
\end{array}$$

The function app_instr expresses the applicability condition for an instruction at the program point pc. Note that besides the conditions formalized in app′, which correspond to the type correctness properties given in Section 2.3, we also require that the stack has a fixed predefined size h in all state types associated with this program point. We employ this requirement to fit the polyvariant model-checking approach into the

[3] This is trivially equivalent to establishing AG Atom φ in a branching-time temporal logic formalization. In other words, as our applicability properties are global invariance properties, our correctness proof holds for both linear and branching-time model checking.

monovariant framework. In order to prove that there is a state type, we must show that the supremum of all possible stack types associated with the program point is different from the error element Err, and hence we impose this restriction.

The function app_method builds φ as the conjunction of the applicability conditions of all program points in the bytecode *ins* (the list *hs* contains, for each instruction in *ins*, a number that specifies the stack size at that program point). The resulting formula expresses the well-typedness condition for a given program state and hence $K \models \Box\,\mathsf{Atom}\,\varphi$ formalizes, for a Kripke structure K resulting from a method abstraction, the global type safety of the method. This formula, together with the additional constraints explained below, makes up the definition of a bytecode model-checked method, which is formalized using the predicate bcm_method.

$$
\begin{array}{l}
\mathsf{bcm\_method} :: \mathit{jvm\_prog} \to \mathit{cname} \to \mathit{ty\ list} \to \mathit{ty} \to \mathit{nat} \to \mathit{nat} \to \\
\qquad\qquad \mathit{instr\ list} \to \mathit{bool} \\
\mathsf{bcm\_method}\ P\ C\ pTs\ rT\ maxl\ maxs\ ins\ \equiv \\
\quad \exists hs.\ \mathsf{let} \\
\qquad\quad K\ =\ \mathsf{abs\_method}\ P\ C\ pTs\ rT\ maxl\ ins; \\
\qquad\quad \varphi\ =\ \mathsf{app\_method}\ P\ hs\ rT\ maxs\ ins \\
\qquad \mathsf{in} \\
\qquad\quad K \models \Box\,\mathsf{Atom}\,\varphi \wedge \\
\qquad\quad \mathsf{size}\ hs\ =\ \mathsf{size}\ ins \wedge \\
\qquad\quad \mathsf{traces}\ (K)\ \neq\ \{\} \wedge \\
\qquad\quad \forall\,(q,\,q') \in (\mathsf{next}\ K).\ \mathsf{reachable}\ K\ q \longrightarrow \mathsf{reachable}\ K\ q') \wedge \\
\qquad\quad 0\ <\ (\mathsf{size}\ ins) \wedge \\
\qquad\quad \mathsf{bounded}\ (\lambda\, n.\, \mathsf{succs}\ (ins!n)\ n)\ (\mathsf{size}\ ins)
\end{array}
$$

The first conjunct in bcm_method expresses that the global applicability conditions for the given bytecode *ins* are checked; in the PROMELA code generated by our system, these properties are expressed in the assert statement (cf. Section 2.4). The second conjunct requires that a stack size is specified for every program point. The third conjunct states that there exists at least one non-trivial model for $\Box\,\mathsf{Atom}\,\varphi$. The fourth conjunct characterizes a progress property of the abstract interpreter. Recall that reachable $K\ q$ means that q lies on an infinite trace. Hence this conjunct says that all branches from infinite traces also lead to infinite traces. The fifth conjunct guarantees that the list of instructions is not empty and the final conjunct states that all instructions have a successor within the method. The properties represented by the first four conjuncts are verified by the model checker, whereas the last two conjuncts, which express static well-formedness properties of the bytecode, are validated by the parser of our bytecode verification system.

7.3. Main theorem and proof

Our main theorem, stating the correctness of bytecode verification by model checking can now be given.

THEOREM 1.

$$(\mathsf{wf\_prog}\ wf\_mb\ P)\ \wedge\ (\mathsf{is\_class}\ P\ C)\ \wedge\ (\forall x \in \mathsf{set}\ pTs.\ \mathsf{is\_type}\ P\ x) \wedge$$
$$(\mathsf{bcm\_method}\ P\ C\ pTs\ rT\ maxl\ maxs\ ins)$$
$$\longrightarrow\ \exists \phi.\ \mathsf{wt\_method}\ P\ C\ pTs\ rT\ maxs\ maxl\ ins\ \phi$$

This theorem states the correctness only for the bytecode *ins* for a single method, of a single class C, of a program P; however, it can easily be extended to a more general statement for the entire program P. We have proved in Isabelle the more general statement but to avoid uninteresting complications, we present the proof of this simpler statement.

The theorem has four assumptions: (A1) the program P is well-formed; (A2) the class C belongs to the program P; (A3) the method signature is well-formed (i.e., all method parameters have declared types); and (A4) the bytecode *ins* has been successfully model checked. Under these assumptions, the conclusion states that *ins* has a well-typing given by the method type ϕ.

More precisely, the conclusion means that the method is not empty and that the start of the method with the method parameters as well as each instruction of the code *ins* is well-typed with respect to ϕ.

To increase readability, in the following we abbreviate $\mathsf{abs\_method}\ P\ C\ pTs\ rT\ maxl\ ins$ as abs_method and $\mathsf{bcm\_method}\ P\ C\ pTs\ rT\ maxl\ maxs\ ins$ as bcm_method.

7.3.1. *Proof intuition*

Our proof of the existence of a method type ϕ is constructive and builds ϕ from the Kripke structure K pointwise for each program point *pc* as follows. First, we define a function $\_\ \mathsf{at}\ \_ :: (\alpha \times \beta)\ kripke \to \alpha \to \beta\ set$, where $K\ \mathsf{at}\ pc \equiv \{\, a \mid \mathsf{reachable}\ K\ (pc, a) \,\}$ denotes the program state types that belong to the program point *pc* reachable on any trace. Second, we build the supremum sup_{pc} of the set K at *pc*. Third, we use the fact (which must be proved) that each supremum sup_{pc} is of the form $\mathsf{OK}\ x_{pc}$ (i.e., it is not Err) and hence x_{pc} is the method type at the program point *pc*. In essence, we perform a dataflow analysis here in our proof to build a method type.

The bulk of our proof establishes that this method type is actually a well-typing for the model-checked bytecode. The main challenge here is to show that each method instruction is well-typed with respect to the instruction's state type (applicability) and that the successors of an

instruction are well-typed with respect to the state type resulting from the execution of the instruction (stability). We establish this by induction on the set K at pc; this corresponds to an induction over all traces that contribute to the set of state types associated with the pcth program point. Since model checking enforces applicability and thus stability for every trace, and since exec is a semi-homomorphism in the sense that

$$\begin{aligned}\mathsf{exec}\,(\bigsqcup_{\mathsf{sl}}\,(\mathsf{OK}\,\text{`}\,\mathsf{Some}\,\text{`}\,(\mathsf{abs\_method\ at}\ pc)))\\ \leq_{\mathsf{le}}\ \bigsqcup_{\mathsf{sl}}\,(\mathsf{exec}\,\text{`}\,(\mathsf{OK}\,\text{`}\,\mathsf{Some}\,\text{`}\,(\mathsf{abs\_method\ at}\ pc)))\,,\end{aligned}$$

it follows that the pointwise supremum of the state types gives rise to a method type that is a well-typing.

7.3.2. *Some proof details*
We now describe the construction of the method type more formally and afterwards the proof that it is a well-typing.

The first two steps of the construction are formalized using the function $\mathsf{ok\_state\_type} :: \mathit{jvm\_prog} \to \mathit{cname} \to \mathit{ty\ list} \to \mathit{ty} \to \mathit{nat} \to \mathit{nat} \to \mathit{instr\ list} \to \mathit{nat} \to \mathit{state}$, where we abbreviate $\bigsqcup_{\mathsf{sl,Err,OK(None)}}$ as $\bigsqcup_{\mathsf{sl}}$.

$$\begin{aligned}&\mathsf{ok\_state\_type}\ P\ C\ pTs\ rT\ maxl\ maxs\ ins\ pc\ \equiv\\ &\quad\bigsqcup_{\mathsf{sl}}\ \mathsf{OK}\,\text{`}\,\mathsf{Some}\,\text{`}\,(\mathsf{abs\_method\ at}\ pc)\end{aligned}$$

Here, the supremum function over sets, $\bigsqcup_{\mathsf{sl}}$, builds from K at pc the supremum sup_{pc} in the semilattice sl. To increase readability, we will use ok_state_type as an abbreviation for ok_state_type $P\ C\ pTs\ rT\ maxl\ maxs\ ins$.

For the third step, we extract the state type using the function $\mathsf{ok\_val} :: \alpha\ err \to \alpha$, where $\mathsf{ok\_val}\ e = \mathtt{case}\ e\ \mathtt{of}\ (\mathsf{OK}\ x) \Rightarrow x \mid \_ \Rightarrow \mathsf{arbitrary}$. Thus the method type at program point pc can be computed by the function $\mathsf{opt\_state\_type} \equiv \mathsf{ok\_val} \circ \mathsf{ok\_state\_type}$ and we chose as the overall method type

$$\phi = \ \mathsf{map}\ (\mathsf{opt\_state\_type}\ P\ C\ pTs\ rT\ maxl\ maxs\ ins)\ [0, \ldots, \mathsf{size}\ ins - 1]\ .$$

Before explaining why ϕ is a well-typing, we state two lemmata central to the proof. As mentioned above, the correctness of our construction requires that for each program point pc, there is a state type x such that $\mathsf{ok\_state\_type}\ pc = \mathsf{OK}\ x$. The following lemma states this, and moreover that the instruction at the program point pc is applicable to data of this type.

LEMMA 5. ***Well-typed supremum***

$$0 < \mathsf{size}\ ins\ \wedge\ pc < \mathsf{size}\ ins\ \wedge\ \mathsf{wf\_prog}\ wf\_mb\ P\ \wedge\ \mathsf{is\_class}\ P\ C\ \wedge$$
$$(\forall x \in \mathsf{set}\ pTs.\ \mathsf{is\_type}\ P\ x)\ \wedge\ \mathsf{bounded}\ (\lambda n.\ \mathsf{succs}\ (ins!n)\ n)\ (\mathsf{size}\ ins)\ \wedge$$
$$(\mathsf{size}\ hs\ =\ \mathsf{size}\ ins)\ \wedge\ (\mathsf{abs\_method} \models \Box\ \mathsf{Atom\ app\_method})$$
$$\longrightarrow \exists x.\ (\mathsf{ok\_state\_type}\ pc = \mathsf{OK}\ x)\ \wedge$$
$$(x = \mathsf{None} \vee (\exists\, st\ reg.\ x = \mathsf{Some}\,(st,\ reg)\ \wedge$$
$$\mathsf{app}'\,(ins!pc,\ P,\ maxs,\ rT,\ (st,\ reg))\ \wedge\ (\mathsf{size}\ st\ =\ hs!pc)))$$

We prove this lemma by induction on the set of program states that belong to the program point pc. This proof uses the fact that if a property holds globally in a Kripke structure then it holds for every reachable state of the structure. The step case requires the following lemma, which states that when two state types (st_1, reg_1) and (st_2, reg_2) satisfy the predicate app' for the instruction $ins!pc$ and the sizes of their stacks are equal, then the supremum of these state types also satisfies the applicability condition for this instruction and the size of supremum's stack is the size of the state types' stacks. Here we abbreviate $\mathsf{lift}\ \mathsf{JType.sup}\ P$ as sup'.

LEMMA 6. ***Well-typed induction step***

$$pc < (\mathsf{size}\ ins)\ \wedge\ \mathsf{semilat\ sl}\ \wedge\ \mathsf{size}\ hs = \mathsf{size}\ ins\ \wedge$$
$$\mathsf{size}\ st_1 = hs!pc\ \wedge\ \mathsf{app}'\,(ins!pc,\ P,\ maxs,\ rT,\ (st_1, reg_1))\ \wedge$$
$$\mathsf{size}\ st_2 = hs!pc\ \wedge\ \mathsf{app}'\,(ins!pc,\ P,\ maxs,\ rT,\ (st_2, reg_2))$$
$$\longrightarrow\ \mathsf{size}\ (st_1 +_{[\mathsf{sup}']}\ st_2) = hs!pc\ \wedge$$
$$\mathsf{app}'\,(ins!pc,\ P,\ maxs,\ rT,\ (st_1 +_{[\mathsf{sup}']}\ st_2,\ reg_1 +_{[\mathsf{sup}']}\ reg_2))$$

We now sketch the proof that ϕ is a well-typing. To simplify notation, we write (st_0, reg_0) as an abbreviation of the initial state type

$$([],\ (\mathsf{OK}\ (\mathsf{Class}\ C))\#(\mathsf{map\ OK}\ pTs)@(\mathsf{replicate}\ maxl\ \mathsf{Err}))\,.$$

Following the definition of wt_method, we must show under the assumptions (A1)–(A4), that three properties hold: First, the method body must contain at least one instruction, i.e. $0 < \mathsf{size}\ ins$. This follows directly from the definition of bcm_method.

Second, the start of the method must be well-typed, that is

$$\mathsf{Some}\,(st_0,\ reg_0)\ \leq_{\mathsf{le\_state\_opt}}\ \phi!0\,.$$

Since the set of traces is not empty, the initial state type $(st_0,\ reg_0)$ is contained in the set abs_method at 0 and hence it follows that

$$\mathsf{OK}\,(\mathsf{Some}\,(st_0, reg_0)) \leq_{\mathsf{le}} \bigsqcup_{\mathsf{sl}}\ (\mathsf{OK}\,`\,\mathsf{Some}\,`\,(\mathsf{abs\_method\ at}\ 0))\,,$$

which is (by the definition of ok_state_type) the same as

$$\mathsf{OK}\,(\mathsf{Some}\,(st_0, reg_0)) \leq_{\mathsf{le}} \mathsf{ok\_state\_type}\ 0.$$

By Lemma 5 we know that the right-hand side of the above inequality is an OK value and thus we can strip off OK yielding $\mathsf{Some}\,(st_0,\ reg_0)\ \leq_{\mathsf{le\_state\_opt}}$ ok_val(ok_state_type 0), which is (by the choice of ϕ and the definition of ok_state_type) the desired result.

Finally, we must show that all instructions of the method *ins* are well-typed, i.e.,

$$\begin{array}{lr}
\forall(pc < \mathsf{size}\ ins).\ \forall\, pc' \in \mathsf{set}\,(\mathsf{succs}\ (ins!pc)\ pc). & \\
\quad pc' < \mathsf{size}\ ins\ \wedge & \text{(boundedness)} \\
\quad \mathsf{app}\ (ins!pc)\ P\ maxs\ rT\ (\phi!pc)\ \wedge & \text{(applicability)} \\
\quad \mathsf{step}\ (ins!pc)\ P\ (\phi!pc)\ \leq_{\mathsf{le_state_opt}}\ (\phi!pc')\,. & \text{(stability)}
\end{array}$$

This splits into three subgoals (boundedness, applicability, and stability). For all three we fix a pc, where $pc <$ size ins and a successor pc' of pc. Boundedness holds trivially, since from bcm_method it follows that succs is bounded.

To show the applicability of the instruction at the program point pc, by the definition of opt_state_type, we must prove

$$\mathsf{app}\ (ins!pc)\ P\ maxs\ rT\ (\mathsf{ok\_val}\ (\mathsf{ok\_state\_type}\ pc))\,,$$

which we establish by case analysis. The first case is when ok_state_type pc is OK None, which in our formalization means that pc is not reachable (i.e. there is dead code). Then applicability holds by the definition of app. The second case is when the Kripke structure abs_method generates program state types that belong to the program point pc. We must then prove

$$\mathsf{app'}\,(ins!pc,\ P,\ maxs,\ rT,\ (st,\ reg))\,,$$

which follows from Lemma 5. This lemma also guarantees that the third case, where ok_state_type pc is Err, does not occur.

The majority of the work is in showing stability. Due to the choice of ϕ and the definition of opt_state_type we must prove

$$\begin{array}{l}
\mathsf{step}\ (ins!pc)\ P\ (\mathsf{ok_val}\ (\mathsf{ok_state_type}\ pc)) \\
\qquad\qquad \leq_{\mathsf{le_state_opt}}\ (\mathsf{ok_val}\ (\mathsf{ok_state_type}\ pc'))\,.
\end{array}$$

By the definition of $\leq_{\mathsf{le}}$ it suffices to show the inequality

$$\begin{array}{l}
\mathsf{OK}(\mathsf{step}\ (ins!pc)\ P\ (\mathsf{ok_val}\ (\mathsf{ok_state_type}\ pc))) \\
\qquad\qquad \leq_{\mathsf{le}}\ \ \mathsf{OK}(\mathsf{ok_val}\ (\mathsf{ok_state_type}\ pc'))\,.
\end{array}$$

Lemma 5 states the applicability of the instruction at program point pc to the state type $\mathsf{ok\_val}\ (\mathsf{ok\_state\_type}\ pc)$ on the left-hand side of the inequality. Hence, by the definition of exec, we can reduce our problem to

$$\mathsf{exec}(\mathsf{OK}(\mathsf{ok\_val}\ (\mathsf{ok\_state\_type}\ pc))) \\ \leq_{\mathsf{le}}\ \mathsf{OK}(\mathsf{ok\_val}\ (\mathsf{ok\_state\_type}\ pc'))\,.$$

Moreover, from Lemma 5 we can also conclude that $\mathsf{ok\_state\_type}$ delivers OK values for pc and pc' and thus the argument of exec is equal to $\mathsf{ok\_state\_type}\ pc$ and the right-hand side of the inequality is equal to $\mathsf{ok\_state\_type}\ pc'$. By expanding the definition of $\mathsf{ok\_state\_type}$, the inequality simplifies to

$$\mathsf{exec}\,(\bigsqcup_{\mathsf{sl}}\,(\mathsf{OK}\,\text{`}\,\mathsf{Some}\,\text{`}\,(\mathsf{abs\_method\ at}\ pc))) \\ \leq_{\mathsf{le}}\ \bigsqcup_{\mathsf{sl}}\,(\mathsf{OK}\,\text{`}\,\mathsf{Some}\,\text{`}\,(\mathsf{abs\_method\ at}\ pc'))\,. \tag{4}$$

In inequality (3) of Lemma 1, we proved that if a function g is a semi-homomorphism and $g\,\text{`}\,A'' \subseteq A'$, then $g\,(\bigsqcup_{\mathsf{sl}}\ A'') \leq_{\mathsf{le}} \bigsqcup_{\mathsf{sl}}\ A'$. Inequality (4) is an instance of the conclusion of this lemma. We can prove that g, here exec, is a semi-homomorphism using Lemma 4. Thus, it suffices to prove that

$$(\mathsf{exec}\,\text{`}\,(\mathsf{OK}\,\text{`}\,\mathsf{Some}\,\text{`}\,(\mathsf{abs\_method\ at}\ pc))) \\ \subseteq\ (\mathsf{OK}\,\text{`}\,\mathsf{Some}\,\text{`}\,(\mathsf{abs\_method\ at}\ pc'))\,.$$

We prove for an arbitrary state type $(st,\ reg) \in \mathsf{abs\_method\ at}\ pc$ that

$$\exists(st',\ reg') \in \mathsf{abs\_method\ at}\ pc'. \\ \mathsf{exec}(\mathsf{OK}(\mathsf{Some}(st,\ reg))) = \mathsf{OK}(\mathsf{Some}\ (st',\ reg'))\,.$$

From $(st,\ reg) \in \mathsf{abs\_method\ at}\ pc$ it follows that $(pc, (st,\ reg))$ is a reachable state of $\mathsf{abs\_method}$, which together with (A4) entails that the applicability conditions hold for $(st,\ reg)$. Hence, by the definition of exec, we can reduce our goal to

$$\exists(st',\ reg') \in \mathsf{abs\_method\ at}\ pc'. \\ \mathsf{OK}(\mathsf{step}\ (ins!pc)\ P\ \mathsf{Some}(st,\ reg)) = \mathsf{OK}(\mathsf{Some}\ (st',\ reg'))$$

and further, by expanding the definition of step and stripping off OK, to

$$\exists(st',\ reg') \in \mathsf{abs\_method\ at}\ pc'.\ \mathsf{step}'\ (ins!pc, P, (st, reg)) = (st', reg')\,.$$

However, this is equivalent to

$$\exists(st',\ reg') \in \mathsf{abs\_method\ at}\ pc'.\ ((st, reg),\ (st', reg')) \in \mathsf{next}(\mathsf{abs\_method})\,,$$

which follows directly from the progress property that is part of (A4).

8. Conclusion

We have given the first comprehensive account of the theory and practice of bytecode verification based on model checking. Based on two different developments we explained how the model-checking approach to bytecode verification works, presented an experimental and theoretical analysis of its complexity, and formally analyzed the correctness of the approach in Isabelle/HOL.

Our experimental analysis constitutes 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, *correct* code can be efficiently validated since only polynomially many states are accessible, provided subroutine nesting and the number of conditional instructions with differently typed branches are 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.

In the formal proof of the correctness of our model-checking approach, we were fortunate in that we could build on the framework of Pusch and Nipkow. As such, our work also constitutes a fairly large-scale example of formal theory reuse, and the generality of their formalism, in particular Nipkow's verification framework, played a major role in this regard. As mentioned in the introduction, the changes we made to the verification framework appear generally useful; our approach supports polyvariant analysis in that it admits bytecode with incompatible types at different program points. However, as discussed in Section 4, for each program point, our formalization requires the stack size to be constant. It would be interesting to lift this requirement in our Isabelle model, for example, by further generalizing the notion of a state type.

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 fully tested and remains as future work. This issue is rather subtle as explained in [9, 14]. 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.

Acknowledgments

We thank Joachim Posegga and Harald Vogt for encouraging us to take up this problem and for their collaboration. The bytecode verifier development was partially supported by Deutsche Telekom AG and T-Nova GmbH. We also thank the anonymous referees for their valuable comments on the complexity analysis of bytecode model checking.

References

1. Basin, D., S. Friedrich, and M. Gawkowski: 2002a, 'Verified Bytecode Model Checkers'. In: V. A. Carreño, C. A. Muñoz, and S. Tahar (eds.): *Theorem Proving in Higher Order Logics (TPHOLs'02)*, Vol. 2410 of *LNCS*. pp. 47–66.
2. Basin, D., S. Friedrich, M. Gawkowski, and J. Posegga: 2002b, 'Bytecode Model Checking: An Experimental Analysis'. In: D. Bosnacki and S. Leue (eds.): *Model Checking Software, 9th International SPIN Workshop*, Vol. 2318 of *LNCS*. pp. 42–59.
3. Basin, D., S. Friedrich, J. Posegga, and H. Vogt: 1999, 'Java Byte Code Verification by Model Checking'. In: N. Halbwachs and D. Peled (eds.): *11th International Conference on Computer-Aided Verification (CAV'99)*. pp. 491–494.
4. Bertot, Y.: 2000, 'A Coq Formalization of a Type Checker for Object Initialization in the Java Virtual Machine'. Research Report RR-4047, INRIA.
5. Biere, A., A. Cimatti, E. Clarke, and Y. Zhu: 1999, 'Symbolic Model Checking without BDDs'. In: W. R. Cleaveland (ed.): *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'99*, Vol. 1579 of *LNCS*. pp. 193–207.
6. Coglio, A.: 2002, 'Simple Verification Technique for Complex Java Bytecode Subroutines'. In: *Proc. 4th ECOOP Workshop on Formal Techniques for Java-like Programs*.
7. Cohen, R.: 1997, 'The Defensive Java Virtual Machine Specification'. Technical report, Computational Logic Inc.
8. Freund, S. N. and J. C. Mitchell: 1999a, 'A Formal Framework for the Java Bytecode Language and Verifier'. *ACM SIGPLAN Notices* **34**(10), 147–166.
9. Freund, S. N. and J. C. Mitchell: 1999b, 'The Type System for Object Initialization in the Java Bytecode Language'. *ACM Transactions on Programming Languages and Systems* **21**(6), 1196–1250.
10. Holzmann, G. J.: 1997, 'The Spin Model Checker'. *IEEE Transactions on Software Engineering* **23**(5), 279–295.
11. Kfoury, A. J., J. Tiuryn, and P. Urzyczyn: 1994, 'An analysis of ML typability'. *Journal of the ACM* **41**(2), 368–398.
12. Klein, G. and T. Nipkow: 2002, 'Verified Bytecode Verifiers'. *Theoretical Computer Science*. 48 pages, to appear.
13. Klein, G. and M. Wildmoser: 2003, 'Verified Bytecode Subroutines'. *Journal of Automated Reasoning*. 38 pages, in this issue.
14. Leroy, X.: 2001, 'Java Bytecode Verification: An Overview'. In: G. Berry, H. Comon, and A. Finkel (eds.): *Computer Aided Verification, 13th International Conference, CAV 2001*, Vol. 2102 of *LNCS*. pp. 265–285.

15. Lindholm, T. and F. Yellin: 1997, *The Java Virtual Machine Specification*, No. 1102 in The Java Series. Addison-Wesley.
16. McMillan, K.: 1992, 'Symbolic Model Checking: An Approach to the State Explosion Problem'. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA. CMU-CS-92-131.
17. Nipkow, T.: 2001, 'Verified Bytecode Verifiers'. In: F. Honsell and M. Miculan (eds.): *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001*, Vol. 2030 of *LNCS*. pp. 347–363.
18. Nipkow, T., D. v. Oheimb, and C. Pusch: 2000, 'μJava: Embedding a Programming Language in a Theorem Prover'. In: F. Bauer and R. Steinbrüggen (eds.): *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*. pp. 117–144.
19. Nipkow, T., L. C. Paulson, and M. Wenzel: 2002, *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*, Vol. 2283 of *LNCS*. Springer.
20. Paulson, L. C.: 1994, *Isabelle: A Generic Theorem Prover; With Contributions by Tobias Nipkow*, Vol. 828 of *LNCS*. Heidelberg: Springer.
21. Posegga, J. and H. Vogt: 1998, 'Byte Code Verification for Java Smart Cards Based on Model Checking'. In: J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann (eds.): *Computer Security – ESORICS 98, 5th European Symposium on Research in Computer Security*, Vol. 1485 of *LNCS*. pp. 175–190.
22. Pusch, C.: 1998, 'Formalizing the Java Virtual Machine in Isabelle/HOL'. Technical Report TUM-I9816, Institut für Informatik, Technische Universität München.
23. Pusch, C.: 1999, 'Proving the Soundness of a Java Bytecode Verifier Specification in Isabelle/HOL'. In: W. R. Cleaveland (ed.): *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'99*, Vol. 1579 of *LNCS*. pp. 89–103.
24. Qian, Z.: 1999, 'A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines'. In: J. Alves-Foss (ed.): *Formal Syntax and Semantics of Java*, Vol. 1523 of *LNCS*. Springer, pp. 271–311.
25. Qian, Z.: 2000, 'Standard Fixpoint Iteration for Java Bytecode Verification'. *ACM Transactions on Programming Languages and Systems* **22**(4), 638–672.
26. Ruys, T. C.: 2001, 'Towards Effective Model Checking'. Ph.D. thesis, University of Twente, Department of Computer Science.
27. Schmidt, D.: 1998, 'Data Flow Analysis is Model Checking of Abstract Interpretations'. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages January 1998, POPL'98*. pp. 38–48.
28. Schmidt, D. and B. Steffen: 1998, 'Program Analysis as Model Checking of Abstract Interpretations'. In: G. Levi (ed.): *Static Analysis, 5th International Symposium, SAS'98*, Vol. 1503 of *LNCS*. pp. 351–380.
29. Stärk, R. F. and J. Schmid: 2001, 'Java Bytecode Verification is not Possible'. In: R. Moreno-Díaz and A. Quesada-Arencibia (eds.): *Formal Methods and Tools for Computer Science, Eurocast*. Extended Abstract.
30. Stata, R. and M. Abadi: 1999, 'A Type System for Java Bytecode Subroutines'. *ACM Transactions on Programming Languages and Systems* **21**(1), 90–137.
31. Yellin, F.: 1995, 'Low Level Security in Java'. In: *World Wide Web Journal: The Fourth International WWW Conference Proceedings*. Cambridge, MA, pp. 369–380.