A Type System for JVM Threads [⋆]

Gaetano Bigliardi and Cosimo Laneve

Department of Computer Science, University of Bologna, Italy.

Abstract. The current definition of the Java Bytecode Verifier, as well as the proposals to formalize it, do not include any check about consistency of critical sections (those between `monitorenter` and `monitorexit` instructions). So code is run, even if critical sections are corrupted. In this paper we isolate a sublanguage of the Java Virtual Machine with thread creation and mutual exclusion. For this subset we define a semantics and a formal verifier that enforces basic properties of threads and critical sections. The verifier integrates well with previous formalizations of the Java Bytecode Verifier. Our analysis of critical sections reveals the presence of bugs in the current compilers from Sun and IBM.

1 Introduction

The Java Programming Language is compiled into an intermediate language, called (Java) *bytecode*. The bytecode is then interpreted by the Java Virtual Machine (JVM, in the following) [1]. Most of the portability of the Java language relies on the fact that JVM's have been defined for almost all the platforms. Indeed, the present scenario is that the bytecode on some machine may be shipped and executed on another one, which is the basis of Java code mobility.

However, bytecode mobility poses a sequel of problems, as bytecodes generated by hostile compilers, or created by attackers, or corrupted during the migration. To contrast possible harmful bytecodes, Java developers have defined the so-called *bytecode verifier*, which checks the format of downloaded classes, the presence of illegal conversions, jumps to invalid addresses, methods invoked with a wrong number or type of actual parameters.

The official definitions of the bytecode verifier consist of a prose description and an implementation [12]. As usual, such definitions are not satisfactory as regards formal reasoning and proof checking. Therefore, very recently, several authors have developed formal specifications of parts of the bytecode verifier, as well as possible extensions [16, 6, 8]. Remarkablely, soundness proofs have been given, and a bug of the Sun implementation of the bytecode verifier has been identified.

There are at least two key features that escape from the present definitions of the bytecode verifier: concurrency and class loading (some progress in the formal

[⋆] This work is partially supported by the Microsoft Research Grant "The Design and the Analysis of Concurrent Object Oriented Languages", the ESPRIT WG-21836 CONFER, and by the Italian MURST Research Program TOSCA.

definition of class loading has been recently done in [15]). In this paper we try to fill the first of them.

At the present time, the bytecode verifier does not filter out the following codes:

```
1  aload_0                 1  aload_0
2  monitorexit             2  monitorenter
3  return                  3  return
```

where, the one on the left releases a lock (`monitorexit` instruction) without having acquired it before, whilst the one on the right acquires the lock (`monitorenter` instruction) without releasing it. As a consequence, the JVM must perform a number of runtime checks in order to rise the following exception:

```
java.lang.IllegalMonitorStateException
```

(the Sun's JDK 1.2 rises no exception for the bytecode on the right).

We supply to these and other deficiencies of the bytecode verifier by designing a type system, thus conforming with previous works of Stata-Abadi and Freund-Mitchell. To this purpose, we define the operational semantics of a fragment of the Java Virtual Machine Language (in the following shortened into JVML) encompassing multithreading and mutual exclusion, and we prove the correctness of our typing system with respect to the operational semantics. Our analysis reveals the presence of bugs in the current compilers from Sun and IBM.

The structure of the paper is the following. Section 2 overviews concurrency in the JVM and in the bytecode, and gives a detailed sketch of what has been achieved and how. Sections 3 and 4 define the syntax and the operational model of JVML_C, the sublanguage of JVML with primitives for thread creation and mutual exclusion. The static semantics is defined in Section 5. In Section 6 we discuss some sound extension of JVML_C with method invocations and with exception handlers. In Section 7 we analyze the bugs we have found in Sun's and IBM's compilers. We comment related works and conclude in Section 8.

2 Threads and mutual exclusion in the bytecode Java

Java supports concurrent programming through *threads* and *monitors* [11]. A new thread of control may be created by (1) creating a (sub-)object of the class `java.lang.Thread` of the standard Java libraries, and then (2) invoking the `start` method of this class. The method `start` spawns a new thread and returns. The control of this new thread is given to the `run` method of the object by the Java Virtual Machine. A thread exits when the `run` method returns. Since the default `run` of `Thread` does nothing, to design a parallel behaviour one must define a sub-class of `Thread` with a new `run` method.

Synchronization across threads is implemented through monitors. That is, each object has an associated lock and synchronization is defined by acquiring and releasing locks. Two forms of synchronization are provided: through *synchronized methods* and through `synchronized` *statements*.

```
public void onlyMe(Foo f) {
  synchronized (f)
    { doSomething(); }
}
```

```
Method void onlyMe(Foo)
  0   aload_1
  1   astore_2
  2   aload_2
  3   monitorenter
  4   aload_0
  5   invokevirtual doSomething()
  8   aload_2
  9   monitorexit
  10  return
  11  aload_2
  12  monitorexit
  13  athrow
  Exception table:
  from to target type
    4   8    11   any
```

Fig. 1. The method `onlyMe` and its sample bytecode of [12], section 7.14

If a thread invokes a synchronized method of an object, the invocation locks the object and releases the lock when the method returns. In between, other invocations of synchronized methods will be blocked. Synchronized methods are usually implemented by the `ACC_SYNCHRONIZED` flag in the constant pool [12]. In particular, the method invocation checks whether the `ACC_SYNCHRONIZED` flag of the method is set. In this case, the lock of the object is implicitly acquired; and will be released when the method returns. Therefore no explicit bytecode instruction is used to this purpose.

The statement `synchronized (x) { S }` models partial synchronization, namely those cases where parts of the method bodies need to be synchronized. Its semantics is to execute the parenthesized statement `S` in a mutual exclusive way on the object `x`, by acquiring and releasing the lock at the beginning and at the end of the execution. A sample compilation of `synchronized` is illustrated in Figure 1. There, the method `onlyMe` acquires the lock of the argument `f`, do something on the self object, then releases the lock and terminates. The instructions 0-1 of the bytecode copy the argument of `onlyMe` in the variable 2. This allows to record in a "safe" place the object that is going to be locked. This "safe" place is always a local variable, that will be called "protected" in the following. The next two instructions attempt to acquire the lock of the argument. The instruction 2 copies the pointer to the protected variable on the stack; the `monitorenter` at 3 is executed provided the object on the stack is unlocked. Once the `monitorenter` terminates, the control is given to the method `doSomething` (instructions 4-5). When `doSomething` returns, the lock is released: at 8 the reference in the protected variable is copied on the stack, and the `monitorexit` at 9 takes this copy to relinquish the lock. In the section from 4 to 9 no other thread may acquire the lock of the protected variable. This section will be called *critical section* in the following. `onlyMe` returns at 10. Lines from 16 to 19 are used to catch exceptions that occur in between instructions from 4 to 8 (see the Exception table). They release the lock and re-bounce the exception to the caller (instruction `athrow`).

The JVM specification is very demanding as regards the implementation of `synchronized` statements by Java compilers. In [12], section 3.11.11, we find the following commitment:

> Proper implementation of synchronized blocks requires cooperation from a compiler targeting the Java virtual machine. The compiler must ensure that at any method invocation completion a `monitorexit` instruction will have been executed for each `monitorenter` instruction executed since the method invocation. This must be the case whether the method invocation completes normally or abruptly [1].

In fact, the scheme described in Figure 1 is faulty, as it is possible for an asynchronous exception ([12], section 2.16.1) to occur between the `aload_2` instruction at 8 and the following `monitorexit`, in which case the monitor may be left in a locked state. Actually, all the present compilers from Sun and IBM do not fulfil the above commitment (see Section 7).

On the other hand, the JVM specification is not so demanding as concerns JVM implementations (see [12], section 8.13). Indeed, such implementations may, but this is not required, enforce the following two rules, guaranteeing *structured locking*. Let t be a thread and ℓ be a lock. Then:

1. The numbers of locks and unlocks performed by t on ℓ during a method invocation must be equal, whether the method invocation completes normally or abruptly.
2. At no point during a method invocation, the number of unlocks performed by t on ℓ may exceed the number of locks performed by t on ℓ.

Needless to say, a JVM missing the structured locking may be exposed to deplored runtime errors, which are notoriously hard to diagnose, such as deadlocks.

In this paper we integrate the Java Bytecode Verifier with static checks of structured locking. To this aim we have singled out the following three properties a bytecode should satisfy:

Property 1. (*Well-formedness*) Instructions `monitorenter` and `monitorexit` of a bytecode must be paired. Every pair has an associated variable, called *protected*, that stores the monitor. The pair `monitorenter/monitorexit` acquires/releases this protected variable (or one alias of its). Instructions executed in between the pair `monitorenter/monitorexit` cannot update the protected variable.

Property 2. (*Correctness of jumps*) Jumps are always inside their own critical section, but outside inner critical sections.

[1] A method invocation completes "abruptly" when its body causes a JVM-exception that is not handled within the method.

Property 3. (*Presence of Exception Handlers*) Every (fragment of) critical section is paired with a piece of code, its *exception handler*, that releases the acquired lock (stored in the protected variable) as well as the locks of the outer critical sections.

Note that Property 1 requires some form of aliasing analysis, since `monitorenter` and `monitorexit` may take a copy of the protected variable, as in the following bytecode:

```
1   aload_0
2   monitorenter
3   aload_0
4   astore_1
5   aload_1
6   monitorexit
```

Once the execution falls inside the critical section starting at line 3, the protected variable 0 is copied into the variable 1. This copy is used at line 5 to load the stack with the protected variable, before exiting from the critical section (line 6). To deal with aliases we use the same technique as in [6] and in the Sun JDK verifier. This technique consists of tracing copies of references by using the line numbers of the instructions duplicating the references. Since we stick to a typed approach, the line numbers will be stored into the types. Furthermore, Property 1 bans critical sections which update the protected variables. This constraint supports exception handlers that, due to an asynchronous exception (see [12], section 2.16.1), are able to release the right lock at every point of the execution of a critical section (Property 4).

Properties 1–3 are implemented by a type system, in the same style of [16] and [6]. Since a thread may modify objects which are in common with other threads, we must carefully check that thread updates of shared data (objects and object locks) do not invalidate the correctness of threads in parallel. To this aim, our model contains the *heap*, a runtime area shared among threads. And typability of a configuration also takes into account the type of objects stored into the heap. The other addition to systems in [16, 6] regards critical sections. To keep track of them, we supplement the type system with block informations, namely a sequence of pairs (i, x) specifying the critical section starting points (i) and the protected variable (x). The type system verifies that every instruction can be properly typed with the block information defining the static nesting of critical sections.

In the first part of the paper, for simplicity, we don't consider exception handlers. Therefore the type system in Section 5 does not assure Property 3. We discuss in Section 6.2 the integration dealing with exception handlers.

3 The syntax of JVML$_C$

The language JVML$_C$ is a restriction of JVML that includes basic constructs and instructions for concurrency. In JVML$_C$, a program is a collection of class declarations:

```
class C {
   super: Thread
   fields: F
   method run ()
    P
}
```

where each class is actually a subclass of `Thread` and only contains the method `run`. Fields F are finite sequences of pairs $a\ \tau$, where $a \in \textsc{Fid}$ is an identifier, and τ is an integer $\textsc{int}$ or an object type $\sigma \in T$. Bodies P are partial maps from *addresses* $\textsc{Addr}$ to instructions. JVML_C has the following instructions:

$$\begin{array}{lcl} \textit{Instruction} & ::= & \texttt{inc} \mid \texttt{pop} \mid \texttt{push0} \mid \texttt{load}\, x \\ & & \mid \texttt{store}\, x \mid \texttt{if}\, L \\ & & \mid \texttt{new}\, \tau \mid \texttt{putfield}\, \sigma.a\ \tau \mid \texttt{getfield}\, \sigma.a\ \tau \\ & & \mid \texttt{start}\, \sigma \mid \texttt{monitorenter}\ x \mid \texttt{monitorexit}\ x \\ & & \mid \texttt{return} \end{array}$$

where x ranges over a finite set of variables $\textsc{Var}$, and L ranges over $\textsc{Addr}$. Variables will be represented by positive integers, but we keep separate the sets $\textsc{Int}$ and $\textsc{Var}$.

The informal meaning of these instructions is as follows:

- `inc` increments the content of the stack; `pop` and `push0` perform the standard operations on the stack; `if` L pops the top value off the stack and either goes through when that value is the integer 0 or jumps to the address L otherwise;
- `new` τ allocates a new object of type τ, initializes it and puts it on top of the stack; `putfield` $\sigma.a\ \tau$ pops the value on the stack and the underlying object value, and assigns the former to the field a of the latter; `getfield` $\sigma.a\ \tau$ pops the object on the stack and pushes the value in the field a;
- `start` σ, `monitorenter` x, and `monitorexit` are the concurrent instructions. The first one creates and starts a new thread for the object on top of the stack. This operation corresponds to

  ```
  invokevirtual java/lang/Thread/start()
  ```

  namely, the standard operation to trigger new threads in the Java bytecode. [2] The instructions `monitorenter` and `monitorexit` are the synchronization primitives that lock and unlock the object on top of the stack. Their arguments are the protected variables;
- `return` terminates program execution.

[2] Unlike the bytecode, our instruction `start` carries an argument, which is the type (or the class name) of the object whose method must be triggered. This is an artifice to define the semantics of `start` : in the JVM the first address of the right method `run` is found in the heap. Here, we prefer to keep the heap as simple as possible and, therefore, we derive the address of `run` from the argument of `start` (see the operational semantics in Figure 3).

The restriction to consider classes as extensions of `Thread` simplifies our analysis, since the method `start` may be safely invoked inside our programs. For classes with constructors and initializers we refer to the analysis in [6] and for classes with other methods see [7]. We assume that initialization is performed *at the same time* of object creation by Java default initializers that put 0 in every integer field and null in every object field.

The JVM instructions `monitorenter` and `monitorexit` take no argument. Indeed, the declaration of the protected variable in these instructions is an expedient to define its static semantics in JVML$_C$. (The protected variable does not play any role in the operational semantics.) Later on, when exceptions will come into the scene, the argument of `monitorenter` and `monitorexit` will become superfluous and it will be dropped, thus recovering the standard syntax (see Section 6.2). Declaring the protected variable in the `monitorenter` and in the `monitorexit` seems annoying because it excludes multiple entry/exit points of the critical section with different objects on the stack (which are not aliases). However, static checkers usually reject such codes because, in general, it is not possible to establish the locked object when the analyzer finds a `monitorexit`.

4 The operational semantics

The bytecode interpreter for JVML$_C$ is defined using the same framework as in [16, 6]. We briefly review the framework before defining our concurrent model.

4.1 Notation, types and values

We start with addresses $\textsc{Addr}$. We assume that $\textsc{Addr}$ and positive integers are different, even if we use the constant 1 and the operation + for the formers. (Addresses will be considered integers in the type system of Section 5.) We write $\mathsf{dom}[P]$ for the set of addresses of P, $P[i]$ being the i-th instruction in P, if $i \in \mathsf{dom}[P]$. If σ is the type of the class of a program P, then we let $1_\sigma \in \mathsf{dom}[P]$, for every P, be the first instruction of P. By extension, when Q is a set of class bodies, we let $1_\sigma \in \mathsf{dom}[Q]$ be the the first instruction of the class body of the class σ.

Partial maps are used to represent most of our entities (heaps, memory functions, etc.). If f is a partial map, we let $f[x \mapsto v]$ be the *updating* operation, that gives the function f where the value of x is v. The symbol ε denotes the empty map (the map undefined everywhere).

Types τ are $\textsc{top}$, integers $\textsc{int}$, object types T and indexed object types $\widehat{T}$. Object types, ranged over by σ, include all the class names of the program. Indexed object types, ranged over by $\widehat{\sigma}$, are defined as follows:

$$\widehat{T} = \{\sigma_i \mid \sigma \in T \ \text{ and } \ i \in \textsc{Addr}\}$$

The type σ_i is used for typing variables whose value of type σ has been copied at line i. Namely, all the variables that have the same type σ_i are aliases.

We define an indexing operation over types τ, in order to mark types when variables are copied. Let τ_i, where $i \in \textsc{Addr}$, be the following:

1. $\tau_i = \tau$, when $\tau = \text{INT}$ or $\tau = \text{TOP}$ (*only copies of references are relevant*);
2. $\tau_i = \sigma_i$, if $\tau = \sigma$ (*the first copy of the variable changes the type*);
3. $\tau_i = \widehat{\sigma}$, if $\tau = \widehat{\sigma}$ (*successive copies of a variable keep the type of the first copy*).

We also let *Ind* be a partial function from types to ADDR defined as follows: $\mathit{Ind}[\sigma_i] = i$; *Ind* is undefined otherwise.

For every σ, we assume a countable set of *object names*, ranged over $o, o', \cdots$. Let $\mathcal{O}$ be the set of all object names. Values v are integer constants or object names. The type TOP includes all the values. The values of types σ and σ_i, for every i, are the same. As usual, we write $v : \tau$ if v is a value of τ.

Finally, for each type σ, we define $\overline{\sigma} = [a_j : \tau_j{}^{j \in J}]$, namely the record of fields that are specified in the corresponding class definition. We address fields with the usual dot notation; therefore, $[a : \tau, a_j : \tau_j{}^{j \in J}].a : \tau$. *Record values* are records $[a_j = v_j{}^{j \in J}]$, where v_j are values.

4.2 The operational model

Each instruction performs a transformation of machine states, that are configurations

$$\Vdash_H \langle pc_1, f_1, s_1, z_1 \rangle, \cdots, \langle pc_n, f_n, s_n, z_n \rangle$$

with the following meaning:

- H is the heap, namely a partial function whose domain is the set $\mathcal{O}$ and whose range is the set of record values. The special field ℓ, $\ell \notin \text{FID}$, represents the *lock* associated with the object. (The instructions `getfield` and `putfield` cannot read/write on the locks.) We assume that record values always have a field ℓ. We use the notation $H(o.a)$ to retrieve the value of the field a in $H(o)$; and the notation $H[o.a \mapsto v]$ to update the field a of o to the value v.
- each tuple $\langle pc_i, f_i, s_i, z_i \rangle$ is a *thread*; pc_i is the address of the instruction to be executed; f_i is a total map from the set VAR of local variables to the set of values; s_i is a stack of values; z_i is a finite subset of $\mathcal{O}$ and represents the set of objects locked by the thread.

Let σ be the type of the class whose `run` is invoked. The machine begins its execution in the state $\Vdash_{H_0} \langle 1_\sigma, f_0[0 \mapsto o], \varepsilon, \emptyset \rangle$, where

- $H_0 = [o \mapsto \rho_\sigma]$, where ρ_σ is the record value of type $\overline{\sigma}$ with fields initialized to 0 and null, according to they are integers or objects. The special field ℓ is initialized to 0.
- f_0 maps the local variables to any values;
- ε is the empty stack.

The rules that define the operational semantics of JVML$_C$ are shown in Figure 2. In Figure 2 we let Q be a set of class bodies, each class body being identified by a different set of addresses. We also leave the set Q implicit in the rewritings. Every rule in Figure 2 actually mentions the components that participate to the

$$\frac{Q[pc] = \texttt{inc}}{\Vdash_H \langle pc, f, n \cdot s, z\rangle \to \Vdash_H \langle pc+1, f, (n+1) \cdot s, z\rangle}$$

$$\frac{Q[pc] = \texttt{push0}}{\Vdash_H \langle pc, f, s, z\rangle \to \Vdash_H \langle pc+1, f, 0 \cdot s, z\rangle} \qquad \frac{Q[pc] = \texttt{pop}}{\Vdash_H \langle pc, f, v \cdot s, z\rangle \to \Vdash_H \langle pc+1, f, s, z\rangle}$$

$$\frac{Q[pc] = \texttt{if}\ L}{\Vdash_H \langle pc, f, 0 \cdot s, z\rangle \to \Vdash_H \langle pc+1, f, s, z\rangle} \qquad \frac{\begin{array}{c}Q[pc] = \texttt{if}\ L\\ n \neq 0\end{array}}{\Vdash_H \langle pc, f, n \cdot s, z\rangle \to \Vdash_H \langle L, f, s, z\rangle}$$

$$\frac{Q[pc] = \texttt{load}\ x}{\Vdash_H \langle pc, f, s, z\rangle \to \Vdash_H \langle pc+1, f, f(x) \cdot s, z\rangle} \qquad \frac{Q[pc] = \texttt{store}\ x}{\Vdash_H \langle pc, f, v \cdot s, z\rangle \to \Vdash_H \langle pc+1, f[x \mapsto v], s, z\rangle}$$

$$\frac{\begin{array}{c}Q[pc] = \texttt{new}\ \sigma\\ o' \notin \mathsf{dom}[H]\\ H' = H[o' \mapsto \rho_\sigma]\end{array}}{\Vdash_H \langle pc, f, s, z\rangle \to \Vdash_{H'} \langle pc+1, f, o' \cdot s, z\rangle}$$

$$\frac{\begin{array}{c}Q[pc] = \texttt{putfield}\ \sigma.a\ \tau\\ H' = H[o.a \mapsto v]\end{array}}{\Vdash_H \langle pc, f, v \cdot o \cdot s, z\rangle \to \Vdash_{H'} \langle pc+1, f, s, z\rangle} \qquad \frac{\begin{array}{c}Q[pc] = \texttt{getfield}\ \sigma.a\ \tau\\ H(o.a) = v\end{array}}{\Vdash_H \langle pc, f, o \cdot s, z\rangle \to \Vdash_H \langle pc+1, f, v \cdot s, z\rangle}$$

$$\frac{\begin{array}{c}Q[pc] = \texttt{start}\ \sigma\\ o \in \mathsf{dom}[H]\end{array}}{\Vdash_H \langle pc, f, o \cdot s, z\rangle \to \Vdash_H \langle pc+1, f, s, z\rangle, \langle 1_\sigma, f_0[0 \mapsto o], \varepsilon, \emptyset\rangle}$$

$$\frac{\begin{array}{c}Q[pc] = \texttt{monitorenter}\ \ x\\ H(o.\ell) = 0\\ H' = H[o.\ell \mapsto 1]\end{array}}{\Vdash_H \langle pc, f, o \cdot s, z \setminus \{o\}\rangle \to \Vdash_{H'} \langle pc+1, f, s, z \cup \{o\}\rangle} \qquad \frac{\begin{array}{c}Q[pc] = \texttt{monitorenter}\ \ x\\ H(o.\ell) = n\ \ (n > 0)\\ H' = H[o.\ell \mapsto n+1]\end{array}}{\Vdash_H \langle pc, f, o \cdot s, z \cup \{o\}\rangle \to \Vdash_{H'} \langle pc+1, f, s, z \cup \{o\}\rangle}$$

$$\frac{\begin{array}{c}Q[pc] = \texttt{monitorexit}\ \ x\\ H(o.\ell) = 1\\ H' = H[o.\ell \mapsto 0]\end{array}}{\Vdash_H \langle pc, f, o \cdot s, z \uplus \{o\}\rangle \to \Vdash_{H'} \langle pc+1, f, s, z\rangle} \qquad \frac{\begin{array}{c}Q[pc] = \texttt{monitorexit}\ \ x\\ H(o.\ell) = n\ \ (n > 1)\\ H' = H[o.\ell \mapsto n-1]\end{array}}{\Vdash_H \langle pc, f, o \cdot s, z \cup \{o\}\rangle \to \Vdash_{H'} \langle pc+1, f, s, z \cup \{o\}\rangle}$$

Fig. 2. The operational semantics of JVML$_C$

rewriting. Of course the rewriting applies to every configuration that contains the components. More explicitly, let $\mathit{Lock}[\mathcal{T}]$ be the collection of objects locked by threads in $\mathcal{T}$, namely $\mathit{Lock}[\mathcal{T}] = \bigcup_{\langle pc,f,s,z\rangle \in \mathcal{T}} z$. Then, for every rewriting rule of Figure 2, we use the rule:

$$\frac{\Vdash_H \mathcal{T}_1 \to \Vdash_{H'} \mathcal{T}_2 \qquad (\mathit{Lock}[\mathcal{T}_1] \cup \mathit{Lock}[\mathcal{T}_2]) \cap \mathit{Lock}[\mathcal{T}] = \emptyset}{\Vdash_H \mathcal{T}_1, \mathcal{T} \to \Vdash_{H'} \mathcal{T}_2, \mathcal{T}} \; \text{(CONTEXT)}$$

where $\mathcal{T}$, $\mathcal{T}_1$ and $\mathcal{T}_2$ are sets of threads. The side-condition $(\mathit{Lock}[\mathcal{T}_1] \cup \mathit{Lock}[\mathcal{T}_2]) \cap \mathit{Lock}[\mathcal{T}] = \emptyset$ bans moves of configurations where two threads locks the same object.

We skip every discussion about the first ten rules of Figure 2, because they are essentially the same as in previous works (see [16, 6]). On the contrary, the last five rules are new and peculiar of our contribution.

The rule modelling the instruction `start` σ creates a new thread of control. The new thread begins into a state where the first instruction of the class σ must be executed, and with the variable 0 containing the object on top of the stack. In particular, none of the locks held by the caller thread is retained by the new thread.

The other rules define the semantics of `monitorenter` and `monitorexit`. The two rules for `monitorenter` state that a thread can acquire a lock of an object if (1) the object is unlocked; or (2) it already owns the lock of that object. Observe that we have sufficient information to establish whether a thread can acquire the lock or wait for it. In particular, the special field ℓ stores the number of times the object has been locked (by the same thread); whilst the field z in the thread collects the set of locks it owns. When the object o is unlocked, `monitorenter` sets the lock field to 1 and stores o in the component z of the thread. In this case, the rule is defined for configurations with z such that $o \notin z$. Alternatively, when the thread holds the object lock, `monitorenter` only increases the lock value. In the two rewriting rules, the argument of `monitorenter` is never used.

The rules for `monitorexit` performs the reverse operations with respect to those of `monitorenter`.

5 The static semantics

In this section we develop the static semantics of JVML$_C$, according to the proposals in [16, 6].

Let Σ be the set of classes of our program Q in JVML$_C$ and let $\{P_\sigma \mid \sigma \in \Sigma\}$ be the collection of bodies therein. We conclude that the program Q is *well-typed*, in notation $F, S, B \vdash Q$, if, for every $\sigma \in \Sigma$, there exist F^σ, S^σ and B^σ such that:

$$F^\sigma, S^\sigma, B^\sigma \vdash P_\sigma \; .$$

The partial maps F^σ and S^σ give the types of local variables, of the stack and of the set of locks when the program points at a given address. The partial map

B^σ defines *critical sections*, that enclose all the instructions between the one following a `monitorenter` and the matching `monitorexit`.

Precisely, these maps have domain ADDR and codomain defined as follows ($i \in$ ADDR):

1. $F^\sigma[i]$ is a map from local variables to types at location i;
2. $S^\sigma[i]$ is a sequence of types of the operand stack at location i;
3. $B^\sigma[i]$ is a sequence $(i_1, x_1) \cdots (i_k, x_k)$, where $i_j \in$ ADDR and $x_j \in$ VAR. These sequences are *well-formed*, namely addresses and variables are pairwise different. We refer to addresses and variables in $B^\sigma[i]$ with $Addr[B^\sigma[i]]$ and $Var[B^\sigma[i]]$, respectively.

We let F^σ_{TOP} be the function that maps 0 to σ and all the other variables to TOP. The application of a partial map G to address i is often abbreviated into G_i.

The rule that proves $F^\sigma, S^\sigma, B^\sigma \vdash P_\sigma$ is:

$$\frac{\begin{array}{c} F^\sigma[1_\sigma] = F^\sigma_{\mathrm{TOP}} \\ S^\sigma[1_\sigma] = \varepsilon \\ B^\sigma[1_\sigma] = \varepsilon \\ \forall i \in \mathsf{dom}[P_\sigma].\ F^\sigma, S^\sigma, B^\sigma, i \vdash P_\sigma \end{array}}{F^\sigma, S^\sigma, B^\sigma \vdash P_\sigma}$$

The top three premises regard the first instruction of the body P_σ. They just set that a new thread starts with a value in the local variable 0 of type σ (see the semantics of `start` σ; whilst the stack and the set of acquired locks are empty. The lowest premise checks that every instruction is well-typed. Figure 3 defines the rules for the judgment $F^\sigma, S^\sigma, B^\sigma, i \vdash P_\sigma$ (in the figure we have always drop the index σ).

To be as much as possible conservative with respect to previous proposals, we have arranged premises of rules in such a way that those in the top conform with the premises of the corresponding rules in [16] and in [6]. The new premises mostly concern the function B. Among these rules, we discuss IF, LOAD and STORE.

To type `if` L at i, one must verify that both the instruction at $i+1$ and that at L can be typed with the same values of F, S and B. This allows to abstract out of the branch that will be taken at run-time. Remark that $B_i = B_{i+1} = B_L$ means that the instructions at i, $i+1$ and L belong to the same critical section. Said otherwise, it is not possible to jump outside a critical section (because the top pair of B_L should have a different address) or inside inner critical sections (because the lengths of B_{i+1} and B_L should be different).

According to LOAD, to verify `load` x at i, the top element of the type stack S_{i+1} must be the type of the variable x. Actually, since `load` x performs a copy of a variable, the type left on the stack is an i-indexed object type. Correspondingly, the type of $F_{i+1}[x]$ is changed, too. Observe that this rule may be applied only

(Inc)
$$\frac{\begin{array}{c} P[i] = \texttt{inc} \\ F_i = F_{i+1} \\ S_i = \textsc{int} \cdot \alpha = S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ B_i = B_{i+1} \end{array}}{F, S, B, i \vdash P}$$

(Push)
$$\frac{\begin{array}{c} P[i] = \texttt{push0} \\ F_i = F_{i+1} \\ \textsc{int} \cdot S_i = S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ B_i = B_{i+1} \end{array}}{F, S, B, i \vdash P}$$

(Pop)
$$\frac{\begin{array}{c} P[i] = \texttt{pop} \\ F_i = F_{i+1} \\ S_i = \tau \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ B_i = B_{i+1} \end{array}}{F, S, B, i \vdash P}$$

(If)
$$\frac{\begin{array}{c} P[i] = \texttt{if } L \\ F_i = F_{i+1} \\ F_L = F_{i+1} \\ S_i = \textsc{int} \cdot S_{i+1} = \textsc{int} \cdot S_L \\ i+1, L \in \mathsf{dom}[P] \\ B_i = B_{i+1} = B_L \end{array}}{F, S, B, i \vdash P}$$

(Load)
$$\frac{\begin{array}{c} P[i] = \texttt{load } x \\ x \in \mathsf{dom}[F_i] \\ F_i[x] = \tau \\ F_i[x \mapsto \tau_i] = F_{i+1} \\ \tau_i \cdot S_i = S_{i+1} \\ \forall \tau \in S_i.\ i \neq \mathit{Ind}[\tau] \\ \forall y \in \mathsf{dom}[F_i].\ i \neq \mathit{Ind}[F_i[y]] \\ i+1 \in \mathsf{dom}[P] \\ B_i = B_{i+1} \end{array}}{F, S, B, i \vdash P}$$

(Store)
$$\frac{\begin{array}{c} P[i] = \texttt{store } x \\ x \in \mathsf{dom}[F_i] \\ F_i[x \mapsto \tau] = F_{i+1} \\ S_i = \tau \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ B_i = B_{i+1} \\ x \notin \mathit{Var}[B_i] \end{array}}{F, S, B, i \vdash P}$$

(New)
$$\frac{\begin{array}{c} P[i] = \texttt{new } \sigma \\ F_i = F_{i+1} \\ \sigma \cdot S_i = S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ B_i = B_{i+1} \end{array}}{F, S, B, i \vdash P}$$

(Putfield)
$$\frac{\begin{array}{c} P[i] = \texttt{putfield } \sigma.a\ \tau \\ \overline{\sigma}.a : \tau \\ F_i = F_{i+1} \\ S_i = \tau \cdot \sigma \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ B_i = B_{i+1} \end{array}}{F, S, B, i \vdash P}$$

(Getfield)
$$\frac{\begin{array}{c} P[i] = \texttt{getfield } \sigma.a\ \tau \\ \overline{\sigma}.a : \tau \\ F_i = F_{i+1} \\ S_i = \sigma \cdot S_i' \\ \tau \cdot S_i' = S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ B_i = B_{i+1} \end{array}}{F, S, B, i \vdash P}$$

(Return)
$$\frac{\begin{array}{c} P[i] = \texttt{return} \\ B_i = \varepsilon \end{array}}{F, S, B, i \vdash P}$$

(Start)
$$\frac{\begin{array}{c} P[i] = \texttt{start } \sigma \\ F_i = F_{i+1} \\ S_i = \sigma \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ B_i = B_{i+1} \end{array}}{F, S, B, i \vdash P}$$

(Monitorenter)
$$\frac{\begin{array}{c} P[i] = \texttt{monitorenter}\ \ x \\ F_i = F_{i+1} \\ F_i[x] = \widehat{\sigma} \\ S_i = \widehat{\sigma} \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ i \notin \mathit{Addr}[B_i] \\ x \notin \mathit{Var}[B_i] \\ B_{i+1} = (i, x) \cdot B_i \end{array}}{F, S, B, i \vdash P}$$

(Monitorexit)
$$\frac{\begin{array}{c} P[i] = \texttt{monitorexit}\ \ x \\ F_i = F_{i+1} \\ F_i[x] = \widehat{\sigma} \\ S_i = \widehat{\sigma} \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ B_i = B' \cdot (i', x) \cdot B'' \\ B_{i+1} = B' \cdot B'' \end{array}}{F, S, B, i \vdash P}$$

Fig. 3. The static semantics of JVML$_C$

if no i-indexed object type already occurs in S_i or F_i. This prevents those run-time situations where two different object values may have the same statically computed indexed type.

The rule STORE verifies that the type stack at $i+1$ is the one at i without the topmost type. This type is recorded into $F_{i+1}[x]$. STORE also verifies that the updated variable is not protected (premise $x \notin \mathit{Var}[B_i]$), namely it has not been used to record a locked object. This is crucial to keep consistent B and for the correctness of our analysis.

The rules START, MONITORENTER and MONITOREXIT are new, and we examine them one by one. Rule START is straightforward because no static check is undertaken for the new thread, that is a run-time entity.

Rules MONITORENTER as well as MONITOREXIT require that the type stack have an indexed object type $\widehat{\sigma}$ at the top. According to MONITORENTER, $\widehat{\sigma}$ must be equal to the type of the protected variable—the argument of `monitorenter`. Missing such argument, it should not be possible to establish this equality. Moreover, MONITORENTER also checks that the variable x has not been used to record the object locked in an outer control section (premise $x \notin \mathit{Var}[B_i]$) and sets the critical section for the next instruction (premise $B_{i+1} = (i, x) \cdot B_i$).

Rule MONITOREXIT requires that the type on top of S_i must coincide with that of the argument (the protected variable). Furthermore, since the next instruction is outside the critical section, MONITOREXIT constraints B_{i+1} to be B_i without the pair (i', x). Note that, by the well-formedness of B_i, there is exactly one pair with the variable x. Again, the argument of the `monitorexit` is essential.

5.1 A program and its typing

To illustrate our type system we compute the type informations for a sample bytecode in Figure 4. This bytecode differs from the body of the method `onlyMe` in Figure 1 in two ways: (1) there is no reference stored in the variable 1; (2) the invocation to `doSomething` has been replaced by an updating of the field `val` of the object stored in the variable 0.

In Figure 4, we have specified the value of F for the variables 0 and 1, since the other variables are always mapped to TOP. The rightmost column defines the value of the function B. Observe that B is not empty in the critical section only (instructions from 5 to 9) and, therein, B keeps the name of variable 1, to forbid possible updates. Remark that $B_{10} = \varepsilon$, this enforces the property that objects locked by the method have been properly unlocked on exiting.

5.2 Main properties

Our main result is the following:

Theorem 1. *Let Q be a well-typed JVML$_C$ program and let $\mathcal{T}$ be a configuration reached during an execution of Q. Then*

i	$P[i]$	$F_i[0]$	$F_i[1]$	S_i	B_i
1	`load` 0	`Sample`	TOP	ϵ	ϵ
2	`store` 1	$\texttt{Sample}_1$	TOP	$\texttt{Sample}_1 \cdot \epsilon$	ϵ
3	`load` 1	$\texttt{Sample}_1$	$\texttt{Sample}_1$	ϵ	ϵ
4	`monitorenter` 1	$\texttt{Sample}_1$	$\texttt{Sample}_1$	$\texttt{Sample}_1 \cdot \epsilon$	ϵ
5	`load` 0	$\texttt{Sample}_1$	$\texttt{Sample}_1$	ϵ	$(4,1) \cdot \epsilon$
6	`push0` 0	$\texttt{Sample}_1$	$\texttt{Sample}_1$	$\texttt{Sample}_1 \cdot \epsilon$	$(4,1) \cdot \epsilon$
7	`putfield` `Agent.ref` INT	$\texttt{Sample}_1$	$\texttt{Sample}_1$	$\text{INT} \cdot \texttt{Sample}_1 \cdot \epsilon$	$(4,1) \cdot \epsilon$
8	`load` 0	$\texttt{Sample}_1$	$\texttt{Sample}_1$	ϵ	$(4,1) \cdot \epsilon$
9	`monitorexit` 1	$\texttt{Sample}_1$	$\texttt{Sample}_1$	$\texttt{Sample}_1 \cdot \epsilon$	$(4,1) \cdot \epsilon$
10	`return`	$\texttt{Sample}_1$	$\texttt{Sample}_1$	ϵ	ϵ

Fig. 4. A program and its static type informations

1. *(Threads are owner of locks they release) Every thread in $\mathcal{T}$ always releases locks that have been previously acquired.*
2. *(Objects are locked by active threads) Every thread in $\mathcal{T}$, upon termination, releases all the locks it acquires.*

The operational semantics in Figure 2 is too rough to directly support the properties of Theorem 1 because of the component z of configurations. Indeed, to verify properties on the ordering of lock releases, we require that z is a stack, rather than a set. The reader can find the proof of Theorem 1 in the full paper [3].

Another important property of the type system in Figure 3 makes possible to recover the structure of the `synchronized`-statement out of well-typed bytecodes with nested critical sections.

Theorem 2. *Let Q be a well-typed $JVML_C$ program where every* `monitorexit` *releases the variable on top of B. It is possible to reorder the instructions of every method in Q in such a way that critical sections are contiguous blocks of instructions starting with a* `monitorenter` *and ending with a* `monitorexit`*, and possibly containing nested critical sections. Jumps are always inside their own critical section and outside inner critical sections.*

The technique works by collecting the fragments of the code which have the same stack in B. Each step may require adjustments of addresses and the insertion of jumps. According to the type system, every block created has exactly one `monitorenter` instruction and may have several corresponding `monitorexit`. Replace all `monitorexit` except the one in the bottom with `goto`-instructions to the remaining `monitorexit`. Finally, blocks are nested one inside the other, according to the structure of B. The well-typing guarantees that jumps are correctly performed.

6 Extensions

In this section we discuss the extensions of JVML$_C$ and its type system we have studied. Section 6.1 deals with method invocations; whilst Section 6.2 is devoted to exception handlers. The reader is referred to the full paper [3] for the analysis of the extension of JVML$_C$ with the three Java primitives to control thread execution: `wait`, `notify` and `notifyAll`.

6.1 Method invocation

The JVM provides four instructions to model method invocation, according to the called method is public (instruction `invokevirtual`) or is implemented by an interface (instruction `invokeinterface`) or is private (instruction `invokespecial`) or is static (instruction `invokestatic`). We discuss the instruction `invokevirtual` (the other ones may be dealt with in similar ways) and, for simplicity, but without loss of generality, we consider void methods with no argument.

Consider the language JVML$_C^m$, that extends JVML$_C$ with the instruction

$$\begin{array}{rl} \mathit{Instruction} ::= & \cdots \text{(as in section 3)} \\ & \mid \mathtt{invokevirtual}\ \sigma.\mathtt{m} \end{array}$$

where $\sigma.\mathtt{m}$ is a *method descriptor*, namely a pair (class-name, method-name) that uniquely identifies a method body. (In general, descriptors also contain method types.) We assume that the initial address of a method $\sigma.\mathtt{m}$ is $1_{\sigma.\mathtt{m}}$.

The formal definition of `invokevirtual` amounts to introduce the concept of *frame*, a memory area that stores data and partial results. More precisely, a frame is a tuple (pc, f, s), where pc is the address of the instruction to be executed, f is the map of local variables and s is the stack. A thread is now a pair $\langle \rho, z \rangle$, where ρ is a nonempty sequence of frames and z is the set of locked objects. The intended meaning is that the control is owned by the initial frame of ρ. It ceases to own the control either when it invokes another method or when it returns. In the former case, a new tuple is added at the beginning of ρ; in the latter case the first frame is removed and the control is given back to the second frame. Notice that all the frames share the same set of locks. (In the JVM, locks are per-thread.)

The rules defining the operational semantics of JVML$_C^m$ are those of Figure 2 patched with frames, plus those in Figure 5. Observe that the instruction `invokevirtual` uses the object on top of the stack of the caller to initialize the variable 0 of the new frame. This is the self-object of the invoked method. Observe also that `return` is unspecified when the length of the sequence of frames is 1.

The static semantics. Since the bytecode verifier checks a program method by method, the analysis of `invokevirtual` does not require any involved test. The inference rule in Figure 6 only verifies that the type of the object on top of the stack matches with the class of the invoked method. (We let Q be the whole program.)

$$\frac{P[pc] = \texttt{invokevirtual}\ \sigma.\mathtt{m}}{\Vdash_H \langle (pc, f, o \cdot s) \cdot \rho, z\rangle \ \rightarrow\ \Vdash_H \langle (1_{\sigma.\mathtt{m}}, f_0[0 \mapsto o], \varepsilon) \cdot (pc, f, o \cdot s) \cdot \rho, z\rangle}$$

$$\frac{P[pc] = \texttt{return}}{\Vdash_H \langle (pc, f, s) \cdot (pc', f', o \cdot s') \cdot \rho, z\rangle \ \rightarrow\ \Vdash_H \langle (pc' + 1, f', s') \cdot \rho, z\rangle}$$

Fig. 5. The operational semantics of JVML$_C^m$

(Invokevirtual)

$$\frac{\begin{array}{c} P[i] = \texttt{invokevirtual}\ \sigma.\mathtt{m} \\ F_i = F_{i+1} \\ S_i = \sigma \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P],\ \ 1_{\sigma.\mathtt{m}} \in \mathsf{dom}[Q] \\ B_i = B_{i+1} \end{array}}{F, S, B, i \vdash P}$$

Fig. 6. The static semantics of `invokevirtual`.

The properties of Theorem 1 are still valid for the language JVML$_C^m$; the details are in [3].

6.2 Exception handlers

When an exception occurs during the execution of a program, the JVM looks for an handler in the corresponding exception table. The exception table is a sequence of tuples ($\mathtt{from}, \mathtt{to}, \mathtt{target}, \mathtt{type}$), where the first three components are addresses and the last one is the type of the exception. So, if the exception occurred at j, the JVM looks for the first tuple ($i, i', i'', \mathtt{type}$) in the exception table such that $i \leq j \leq i'$ and the type of the exception fits with `type`. In this case the control is given to the instruction at i'', otherwise the exception is rethrown to the caller. If no handler is found for that exception, the program terminates (in an abrupt way).

Actually, the tuple ($i, i', i'', \mathtt{type}$) is *inadequate* with respect to the *structured locking* in Section 2. In particular, to implement structured locking, entries of the exception table that concern critical sections should also specify the protected variable that have to be released. Therefore, in the following, we will assume that entries of the exception table may also have the shape ($i, i', i'', \mathtt{type}, x$), where x is the protected variable.

We remark that the protected variable in ($i, i', i'', \mathtt{type}, x$) is superfluous in the outputs of current compilers. In such bytecodes, it is possible to deduce this variable from the structure of exception handlers of critical sections. For instance, in the Sun's JDK 1.3, the protected variable is always the argument

$$\frac{\begin{array}{c} Q[pc_1] = \texttt{athrow} \\ (pc_i \notin \mathsf{dom}[E])^{i \in 1..n-1} \\ E[pc_n] = (j, j', k) \end{array}}{\Vdash_H \langle (pc_1, f_1, o_1 \cdot s_1) \cdots (pc_n, f_n, o_n \cdot s_n) \cdot \rho, z \rangle \ \rightarrow \ \Vdash_H \langle (k, f_n, o_1) \cdot \rho, z \rangle}$$

Fig. 7. The operational semantics of `athrow`

of the second instruction of the exception handler (a `load` instruction—see Section 7).

To model the type of exceptions, we augment the set of object types T with the type THROWABLE. For simplicity, we drop the last component of entries of exception tables (the `type` component), assuming that the type is always THROWABLE.

We also assume that a method body is now a pair $(P\,;E)$, where E is the *exception table*, namely a partial map from $\mathsf{dom}[P]$ to $\mathsf{dom}[P] \times \mathsf{dom}[P] \times \mathsf{dom}[P]$ or to $\mathsf{dom}[P] \times \mathsf{dom}[P] \times \mathsf{dom}[P] \times \textsc{Var}$. $E[pc]$ gives the first tuple (i, i', k, x) or (i, i', k) in the exception table E such that $i \leq pc \leq i'$. By extension, E will also denote the function that is the union of the exception tables in every method of the program Q. Finally, exceptions are supposed to be thrown only by instructions `athrow`.

Let JVML_C^e be the following extension of JVML_C^m:

$$\begin{array}{rl} \textit{Instruction} ::= & \cdots \text{(as in section 6.1 without } \texttt{monitorenter}\ x \text{ and } \texttt{monitorexit}\ x\text{)} \\ & \mid \texttt{monitorenter} \ \mid \texttt{monitorexit} \\ & \mid \texttt{athrow} \end{array}$$

Remarkablely, the argument of `monitorenter` and `monitorexit`—the protected variable—has been dropped, thus retrieving the standard syntax of the JVM. As we will see, the type system will retrieve the protected variable from the exception table entry. The operational semantics of `athrow` is defined in Figure 7. Observe that, when an exception is thrown, the JVM looks for the first frame which has an entry in the exception table. If such a frame is found, the control is given to the corresponding handler that begins with a stack containing the exception. All the previous frames are deleted.

Static semantics. To type exception tables, we begin by refining the notion of well-typed program (see also [7] for a similar development). Let Σ be the set of classes of the program Q and let $\{(P_\sigma; E_\sigma) \mid \sigma \in \Sigma\}$ be the collection of bodies therein. The program Q is well-typed, in notation $F, S, B \vdash Q$, if, for every $\sigma \in \Sigma$, there exist F^σ, S^σ and B^σ such that:

$$F^\sigma, S^\sigma, B^\sigma \vdash (P_\sigma; E_\sigma)\ .$$

$$\text{(ATHROW)}\quad \frac{\begin{array}{c} P[i] = \texttt{athrow} \\ S_i = \text{THROWABLE} \cdot S' \\ B_i \neq \varepsilon \;\Rightarrow\; \begin{cases} E[i] = (j, j', k, x) \\ B_i = B_k \end{cases} \end{array}}{F, S, B, i \vdash (P; E)}$$

$$\text{(MONITORENTER-EXC)}\quad \frac{\begin{array}{c} P[i] = \texttt{monitorenter} \\ F_i = F_{i+1} \\ F_i[x] = \widehat{\sigma} \\ S_i = \widehat{\sigma} \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ i \notin \mathit{Addr}[B_i] \\ x \notin \mathit{Var}[B_i] \\ B_{i+1} = (i, x) \cdot B_i \\ E[i+1] = (i+1, i', k, x) \end{array}}{F, S, B, i \vdash (P; E)}$$

$$\text{(MONITOREXIT-EXC)}\quad \frac{\begin{array}{c} P[i] = \texttt{monitorexit} \\ F_i = F_{i+1} \\ F_i[x] = \widehat{\sigma} \\ S_i = \widehat{\sigma} \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ B_i = B' \cdot (i', x) \cdot B'' \\ B_{i+1} = B' \cdot B'' \\ E[i] = (i', i, k, x) \end{array}}{F, S, B, i \vdash P}$$

Fig. 8. The static semantics of `athrow` and (the refinement of) MONITORENTER.

and for every tuple $\xi \in E[\mathsf{dom}[P]]$, $F^\sigma, S^\sigma, B^\sigma \vdash \xi$ handles P_σ. This last judgment is defined by the rule:

$$\frac{\begin{array}{c} i, i', k \in \mathsf{dom}[P_\sigma] \\ S_k = \text{THROWABLE} \\ B_i = B_k \\ \forall y \in \mathit{Var}[B_i].\, F_i[y] = F_k[y] \\ \forall y \notin \mathit{Var}[B_i].\, F_i[y] = \text{TOP} \\ B_i = (i'', x) \cdot B' \end{array}}{F^\sigma, S^\sigma, B^\sigma \vdash (i, i', k, x) \text{ handles } P_\sigma}$$

(the rule for tuples (i, i', k) misses the premise in the bottom). The above rule is more restrictive with respect to rule (*wt handler*) in [7]. The reason is twofold. Firstly, we have a rudimentary form of subtyping polymorphism, namely TOP is the type of every value. Variables that are not used by the handler are given type TOP, rather than a suitable super-type. Secondly, it is not possible to clean variables that are protected when the exception is thrown because they will be used by the handler to unlock the right objects. For this reason we also impose the constraint $B_i = B_k$.

Figure 8 defines the rule for the static correctness of `athrow` and a refinement of MONITORENTER and MONITOREXIT. Rule ATHROW checks that the object on top of the stack is of type THROWABLE and the existence of an handler if the instruction `athrow` occurs inside a critical section. In this case, both the handler and the address of `athrow` must have the same value of B. Indeed, operationally, `athrow` is a jump and this constraint is similar to one found in the rule IF.

Rule MONITORENTER-EXC adds one constraint to MONITORENTER requiring a suitable entry of the exception table to handle exceptions occurring in the critical section. This premise is crucial in our verifier. Indeed, $E[i+1] = (i+1, i', k, x)$, in combination with the rule for typing entries of the exception table, guarantee that the initial instruction of the handler is typed with the same value of B of the critical section. Consequently, by rule Return, a well-typed handler may return provided its code releases every lock in the block information. Similarly, Monitorexit-exc adds the constraint $E[i] = (i', i, k, x)$ to Monitorexit. This constraint allows to retrieve the protected variable of the critical section.

Therefore, we may conclude with the following property (that yields a weak form of Property 3 in Section 2).

Theorem 3. *Let Q be a well-typed $JVML_C^e$ program and let $\mathcal{T}$ be a configuration reached during an execution of Q. If exceptions are due to* `athrow` *instructions then $\mathcal{T}$, upon termination, always releases every lock that has acquired.*

This theorem gives no guarantee against exceptions that are not raised by `athrow` instructions. To extend this result, the verifier should check that every line i, which is typed with a not empty B_i, is protected by an exception handler whose initial instruction has the same B_i. We detail this question in [3]; Section 7 contains a bytecode that fits with the above extension.

7 Compilation of `synchronized` in current Java compilers

In this section we analyze the outputs of three Java compilers: the Sun's JDK 1.2 and 1.3 and the IBM's Jikes. As specified in [12] these compilers should conform to the commitment in Section 2. As we will see, all of them definitely fail this goal.

Consider the method `onlyMe` in Figure 1. Figure 9 illustrates the outputs of JDK 1.2 and 1.3. Note that the output of JDK 1.2 is almost the same as the bytecode compiling the `onlyMe` method in Figure 1 (that is taken from [12], section 7.14): the difference is not meaningful for the following discussion.

While the JDK 1.2 output is correct if exceptions are only raised by `athrow` statements, it is faulty if asynchronous exceptions may occur (see [12], section 2.16.1). In particular, it is possible for an asynchronous exception to occur between the `aload_2` instruction at location 8 (or at 13) and the following `monitorexit`. In this case the monitor may be left in a locked state because the exception table does not protect instructions at 9, 13 and 14.

To remedy to this problem, the designers have chenged the compiler in the JDK 1.3 version, as illustrated by the output in Figure 9. However, the solution proposed by this bytecode is still bugged. Very strangely, the block protected by the exception handler is widened from 4 to 13 (and the exception handler begins at 13 itself). Therefore, if an exception is raised at line 10, the lock of the object released at line 9 will be released a second time at line 14, thus causing an exception `IllegalMonitorStateException`. Furthermore, the exception handler itself is compiled in a different way. In JDK 1.2, the lock is immediately released by the exception handler and the exception is thrown again. In JDK 1.3, the first instruction of the exception handler stores the exception in a variable. Afterwards, this variable is reloaded and the exception thrown again. This different scheme is not clear: it seems as the same work is done using 5 instructions rather than 3.

```
Method void onlyMe(Foo)
  0   aload_1
  1   astore_2
  2   aload_2
  3   monitorenter
  4   aload_0
  5   invokevirtual doSomething()
  8   aload_2
  9   monitorexit
  10  goto 16
  13  aload_2
  14  monitorexit
  15  athrow
  16  return
  Exception table:
  from to target type
    4   8    13   any

     JDK 1.2
```

```
Method void onlyMe(Foo)
  0   aload_1
  1   astore_2
  2   aload_2
  3   monitorenter
  4   aload_0
  5   invokevirtual doSomething()
  8   aload_2
  9   monitorexit
  10  goto 18
  13  astore_3
  14  aload_2
  15  monitorexit
  16  aload_3
  17  athrow
  18  return
  Exception table:
  from to target type
    4  13   13   any

     JDK 1.3
```

Fig. 9. Compilations of the `synchronized` statement by Sun's JDK 1.2 and JDK 1.3

The IBM Jikes 1.11 compiler follows the same pattern of JDK 1.3, with a fragment of code at the end which seems unreachable. Therefore Jikes 1.11 shares the same criticisms with JDK 1.3.

7.1 Self-protecting exception handlers

We have reported our analysis to Sun and IBM developers. Sun developers will repair JDK 1.3 in the next version 1.4 in order to solve the above problems. At the present, we are not aware of their solution. Here we illustrate a correct (with respect to our type system) solution.

The common problem in the bytecodes of Figure 9 is the menagement of asynchronous exceptions raised when the interpreter is running the code of the exception handler. For example, if the exception occurs at line 14 of the two bytecodes, the flow of the computation is deviated to the caller, thus preventing any release of the lock. Therefore a correct exception table for the JDK 1.3 output should be:

```
Exception table:
  from to target type
   4    9   13   any
   13  15   13   any
```

The reader may observe that the protected block of the critical section goes, correctly, from 4 to 9. In fact, these are the instructions where the lock acquired at line 3 is helded. However, the main difference from other bytecodes is the presence of a further entry in the exception table. This entry guarantees a "self-protection" to the exception handler: if an exception is raised while the exception handler is running and the handler has still not released the lock then the exception must be managed by the handler itself. In our case, if an exception is raised from 13 to 15, the old exception is discarded and the new exception is handled in the usual way at line 13.

The above solution seems at odd with Sun's compilers, which generate code without "self-protecting" exception handlers (see [12], section 4.9.5).

8 Related works and concluding remarks

As already remarked, our work is strongly based on the framework developed in a series of papers by Stata-Abadi [16] and Freund-Mitchell [6, 7, 8], with the admitted aim of covering most of the static analysis problems of JVML. Other approaches to bytecode verification, that don't cover concurrency issues, are based on data flow analysis [9], typed assembly languages [13] and the Haskell type checker [17].

As regards the bytecode, a very detailed semantics can be found in Bertelsen's works [2]. However Bertelsen does not address the semantics of multi-threading, as well as that of `monitorenter` and `monitorexit` (in his work these instructions have been regarded with the same semantics of `pop`). Another formal semantics of a sublanguage of JVML has been independently defined by Qian [14]. Also Qian misses the concurrent fragment.

Moving away from the bytecode, other works that share the same approach about static analysis concern the Java language (see [4] and the references therein). To be fair, we admit that most of the problems addressed in this paper disappear in the high-level language. Because the `synchronization` statement explicitly defines the critical section and the locked object. Therefore, the integration of the previous works with the concurrent primitives should not be difficult. Nevertheless, this comment does not weaken at all the results of the present paper. What makes the Java language a distinguished programming language is that its bytecode may be transmitted across different machines. A security layer—the bytecode verifier—is needed to safeguard machines from executing hostile bytecodes.

There are two kinds of extensions that have not been considered yet.

The first one concerns the integration of our verifier with other features of the JVM, in the same style of Section 6. Among the others, subroutines seem problematic because they require a form of polymorphism on local variables that are not used therein. We are confident that methods already developed in [16, 8] should be easily integrated inside our verifier.

The second kind of extensions concerns behavioural properties (i.e. safety and liveness properties, see Chapters 2 and 3 of [11]). To this respect, the studies for detecting race conditions among threads in [5] and those about deadlock freeness [10] provide a source of inspiration because they strongly rely on type systems. In any case, the formal model defined here should be a ground basis for every verifier aiming at checking concurrent properties.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[2] P. Bertelsen. Semantics of Java byte code. Technical report, Department of Information Technology, Technical University of Denmark, March 1997.

[3] G. Bigliardi and C. Laneve. A type system for jvm threads. Technical Report UBCLS 2000-06, Dept. of Computer Science, University of Bologna, July 2000.

[4] S. Drossopoulou and S. Eisenbach. Java is type safe — probably. In *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 389–418. Springer-Verlag, 1997.

[5] C. Flanagan and M. Abadi. Types for safe locking. *Lecture Notes in Computer Science*, 1576:91–108, 1999.

[6] S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 310–328, 1998.

[7] S. N. Freund and J. C. Mitchell. A formal framework for the Java bytecode language and verifier. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 147–166, 1999.

[8] S. N. Freund and J. C. Mitchell. A type system for java bytecode subroutines and exceptions. Technical Report STAN-CS-TN-99-91, Stanford Computer Science Technical Note, August 1999.

[9] M. Hagiya and A. Tozawa. On a new method for dataflow analysis of Java Virtual Machine subroutines. *Lecture Notes in Computer Science*, 1503:17–32, 1998.

[10] N. Kobayashi. A partially deadlock-free typed process calculus. In *Proceedings, Twelth Annual IEEE Symposium on Logic in Computer Science*, pages 128–139. IEEE Computer Society Press, 1997.

[11] D. Lea. *Concurrent programming in Java: design principles and patterns.* The Java series. Addison-Wesley, 1996.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison Wesley Longman, Inc., second edition, 1999.

[13] R. O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Conference Record of POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 70–78. ACM Press, 1999.

[14] Z. Qian. Formal specification of a large subset of Java virtual machine instructions for objects, methods and subroutines. In *Formal Syntax and Semantics of Java*, Lecture Notes in Computer Science. Springer-Verlag, Berlin Germany, 1998.

[15] Z. Qian, A. Goldberg, and A. Goglio. A Formal Specification of Java Class Loading. Technical report, Kestrel Institute, Palo Alto, March 2000.

[16] R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.

[17] P. M. Yelland. A compositional account of the Java virtual machine. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 57–69, 1999.