BicolanoMT: a Formalization of Multi-Threaded Java at Bytecode Level [1]

Gustavo Petri [a,2] Marieke Huisman [a,3]

[a] *INRIA Sophia Antipolis*
2004, route des Lucioles BP 93
06902 Sophia Antipolis, France

Abstract

This paper describes a formalization of multi-threaded Java bytecode in Coq. The formalization builds on the existing Bicolano formalization for sequential Java bytecode – which captures basically all aspects of sequential bytecode supported by the CLDC (Java for mobile phones) platform. We use a special extension framework to extend the existing formalization in a systematic way. The formalization is complete: it models all aspects related to concurrency: monitors, thread start and completion, the wait-notify mechanism and the interrupt mechanism, and it does not require any transformation of the bytecode. The formalization is developed to be suited for program verification and static analysis.

1 Introduction

Formal techniques are often advocated as the best way to ensure software security. However, before using such a technique, one needs to establish its correctness formally. This requires a formalization of the underlying programming language's semantics. Ideally, such a semantics is formalized in the logic of a theorem prover, to support the correctness proof of the formal technique.

As a result, many different language formalizations exist (see for example [1,12,13,10]). However, most of these formalizations do not consider concurrency aspects, even though, due to their complexity, the need for a formal foundation is even bigger for concurrent programs. This paper presents a formalization of a multi-threaded language: we extend the sequential bytecode formalization Bicolano [5] with concurrency primitives, as they exist in Java.

[1] This work is partially funded by the IST programme of the EC, under the IST-FET-2005-015905 Mobius project, and the French national research organization (ANR), under the ANR-06-SETIN-010 ParSec project.

[2] Email: gustavo.petri@sophia.inria.fr

[3] Email: marieke.huisman@sophia.inria.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Sequential Bicolano is a Coq formalization of Java bytecode, restricted to the subset of Java that is used for MIDP applications on the CLDC platform [23]. It is used as the formal foundation for the Proof Carrying Code framework, developed in the Mobius project [4]. The Bicolano formalization has been used to prove soundness of a program logic and a verification condition generator for Java [5]. Moreover, an extension for information flow types exists, that has been used to show soundness of an information flow type system [2].

This paper presents the extension of Bicolano towards multi-threading. In particular, we add instructions `monitorenter` and `monitorexit`, and we provide a semantics for the (non-deprecated) native methods related to concurrency (`start`, `join`, `wait`, `notify` and `notifyAll`). Moreover, we model the interrupt mechanism of Java. We are aware of a few other formalizations of multi-threaded Java (discussed in Section 6). Compared to those, we use a similar structure to extend the state of a sequential program to the multi-threaded setting, but we would like to stress the following points that we believe distinguish our formalization from the others:

- Our formalization is faithful to Java, *i.e.*, a bytecode class can be directly mapped into the Coq representation that we use, and no artificial (and hopefully semantics-preserving) code transformations – such as wrapping the body of a synchronized method in a synchronized statement block – are necessary. Currently, the tool `bico` that generates Coq files from bytecode classes is under development (see [6, Ch. 6] for a short description).
- Since we describe the Java semantics at bytecode level, our formalization has the right level of granularity to handle multi-threading, *i.e.*, interleaving of sequential instructions is described naturally. Moreover, in contrast to source code level formalizations (like [16]), we do not have to add special tags to mark that execution is within a synchronized block.
- We build our formalization as an extension of the sequential Bicolano framework in a systematic way, using the extension technique proposed by Czarnik and Schubert [8]. Sequential Bicolano is a fairly complete formalization of Java bytecode, and we thus get this specification "for free". Moreover, because of the use of the extension framework, any extensions made in the sequential setting carry over immediately to the multi-threaded setting.
- Sequential Bicolano has been used to show soundness of program logics and type systems for secure information flow. Our extension for multi-threading allows to extend these soundness results for multi-threaded programs (*e.g.*, an extension of the program logic with rely-guarantee, or a formalization of a type system for secure information flow of multi-threaded programs [3], where the sequential part is already formalized [2]). Moreover, using Coq's extraction mechanism, we can get the implementation of a verified verifica-

[4] See `http://mobius.inria.fr` for more information.

tion condition generator or type checker for free.

- In earlier work, we have formalized the Java Memory Model [11]. For our current formalization, we do not take the Java Memory Model into account, and we simply assume an interleaving semantics (since the Data Race Freeness guarantee of the Java Memory Model ensures that any program without data races only has sequentially consistent behaviors). However, we plan to connect the two formalizations, and to prove that for data race free programs, any BicolanoMT execution can be mapped into a Java Memory Model execution, and vice versa.

The remainder of this paper is organized as follows. Section 2 gives a very brief summary of multi-threading in Java. Next, we discuss relevant aspects of sequential Bicolano and the extension framework. Section 5 is the main part of the paper: it describes how we formalize the multi-threaded features of Java bytecode. Finally, Sections 6 and 7 discuss related work and conclude.

2 Multi-Threading in Java

Concurrency in Java is achieved through multiple threads of execution that communicate via a single shared memory. We list the relevant instructions and (native) methods. For a concise, but more detailed description of concurrency in Java, we refer to Lea [14, Ch. 1]. The class `Thread` represents the actual threads in the Java language. It declares (native) methods to create and handle threads: `start` (spawn a thread for a given thread instance), `join` (wait for a thread to die), and `interrupt` (interrupt a thread).

Because of the use of single shared memory, synchronization is needed to avoid data races, *i.e.*, different threads trying to access the same part of memory simultaneously. This is important, because data races can cause unexpected behavior. The most basic form of synchronization in Java is its built-in locking mechanism, provided by instructions `monitorenter` and `monitorexit`, and by synchronized methods. In addition, Java provides a wait-notify mechanism (implemented by native methods `wait`, `notify` and `notifyAll`), where threads can wait for a certain condition to hold, and other threads can notify one or all waiting threads that this condition might have been established.

3 Sequential Bicolano

Bicolano is a formalization of the Java Virtual Machine Specification developed by D. Pichardie and others [21], using the Coq proof assistant [7], in the context of the Mobius project. Because of its original focus on MIDP applications (*e.g.*, for mobile phones), Bicolano considers a limited subset of the Java language. For example, it does not formalize user-defined class loaders or reflection, and in particular it does not consider multi-threading.

To guarantee that changes in (or extensions of) sequential Bicolano are

propagated directly to the multi-threaded extension of Bicolano, we have build this on top of the original Bicolano framework, using the extension mechanism proposed by Czarnik and Schubert [8], as discussed in the next section. This paper only briefly discusses the most important features of sequential Bicolano, for more information we refer to [5].

Bicolano contains an axiomatic base and a description of the semantics. The axiomatic base defines the abstract syntax, *i.e.*, types of classes, methods, instructions *etc.* with corresponding operations, and the semantic domain, that formalizes the different data structures present in the JVM [15], such as the heap, and the callstack. To show consistency of the axiomatizations, Bicolano also contains example instantiations. The semantics is described as an operational semantics [5]. We discuss the axiomatization of some of the relevant JVM data structures, and a fragment of the small step operational semantics.

JVM Data Structures

The *heap* is modeled as an abstract map, with operations `get`, `update`, `typeof` and `new`. Figure 1 contains its Coq formalization. The argument of the `get` and `update` operations is of type `AddressingMode`: it contains all the information needed to access a field in the heap, such as the `FieldSignature` for static fields, an additional `Location` for dynamic fields, and an offset for array elements. Not all addresses in the domain of `Location` do actually contain a value, therefore `get` and `typeof` are partial functions, modeled using the `option` data type. The `new` function returns a free `Location` in the heap of the type specified by the `LocationType` parameter, while modifying the heap with type information about the new location. Since the `new` function could be undefined for certain `LocationType` values, it is a partial function. The behavior of the different operations is axiomatized via assumptions on the module's operations. Every instantiation of the module should prove that these assumptions are satisfied. Figure 1 contains a few examples of such assumptions, specifying how the `get` and `update` operations interact (where the predicate `Compat` checks the compatibility of an address and the heap).

The *frame* data structure contains the information related to a method call: method description, current program counter, operand stack and values of the local variables. The *callstack* is defined as a list of frames, denoting the unfinished method calls.

```
Module Type FRAME.
 Inductive t : Type :=
   make : Method -> PC -> OperandStack.t -> LocalVar.t -> t.
End FRAME.
```

Finally, the complete JVM state is modeled by the *State* data structure.

[5] Sequential Bicolano contains both a small step and a big step operational semantics – our semantics is built on top of the small step operational semantics, as this is more natural for multi-threading.

```
Module Type HEAP.
 Parameter t : Type.
 Parameter get : t -> AddressingMode -> option value.
 Parameter update : t -> AddressingMode -> value -> t.
 Parameter typeof : t -> Location -> option LocationType.
 Parameter new : t -> Program -> LocationType -> option (Location * t).
 Axiom get_update_same : forall h am v,
   Compat h am -> get (update h am v) am = Some v.
 Axiom get_update_old : forall h am1 am2 v,
   am1<>am2 -> get (update h am1 v) am2 = get h am2.
 Axiom get_uncompat : forall h am, ~Compat h am -> get h am = None.
End HEAP.
```

Fig. 1. Fragment of module `HEAP`

```
Inductive step (p:Program) : State.t -> State.t -> Prop :=
| putfield_step_ok : forall h m pc pc' s l sf f loc cn v,
  instructionAt m pc = Some (Putfield f) ->
  next m pc = Some pc' ->
  Heap.typeof h loc = Some (Heap.LocationObject cn) ->
  defined_field p cn f ->
  assign_compatible p h v (FIELDSIGNATURE.type (snd f)) ->
  step p (St h (Fr m pc (v::(Ref loc)::s) l) sf)
    (St (Heap.update h (Heap.DynamicField loc f) v) (Fr m pc' s l) sf)
```

Fig. 2. Fragment of `step` relation

The state is either normal, or exceptional. In both cases, the state contains a heap and a callstack, however the exceptional frame present in the exceptional state does not contain an operand stack, but only a single location (containing a reference to an exception).

```
Module Type STATE.
 Inductive t : Type :=
   normal : Heap.t -> Frame.t -> CallStack.t -> t
 | exception : Heap.t -> ExceptionFrame.t -> CallStack.t -> t.
End STATE.
```

Operational Semantics

The operational semantics is described by the inductively defined `step` relation. For each JVM instruction, one or more cases describe how the state is changed by the instruction – in particular if an instruction might throw a run-time exception, extra cases specify under which conditions these exceptions occur. Notice that in sequential Bicolano, the `step` predicate is deterministic, *i.e.*, there is always only one case that applies. For illustration, Figure 2 shows a fragment of the `step` relation, specifying the effect of the `putfield` instruction – in case it does not throw any exceptions.

```
Module Type SS_SEM.
 Declare Module Dom: SEMANTIC_DOMAIN.
 Import Dom Dom.Prog.
 Parameter state_t : Type.
 Definition bottom_state_t := State.t.
 Parameter get_bottom_state: state_t -> bottom_state_t -> Prop.
 Parameter step: Program -> state_t -> state_t -> Prop.
 Parameter additional_step: Program -> state_t -> bottom_state_t ->
                            state_t -> Prop.
 Axiom add_step_step_compat: forall p st1 st2 bost2,
   step p st1 st2 -> get_bottom_state st2 bost2 ->
                     additional_step p st1 bost2 st2.
End SS_SEM.
```

Fig. 3. Signature for vertical extension

4 Extensional Framework for Bicolano

As mentioned above, Czarnik and Schubert [8] propose a framework to systematically extend the semantics of Bicolano with additional information and/or additional behavior. They define two kinds of extensions: a *horizontal extension* adds additional information (*e.g.*, resource consumption) to the semantics, but does not change it, while a *vertical extension* may modify the definition of state, and the step relation, provided that the added ingredients are coherent with the underlying semantics. A typical example is to limit the memory size, and to add the possibility to throw an `OutOfMemoryError`. Extensions are extendable themselves, *i.e.*, they can be built on top of each other. Thus our extension with multi-threading can be stacked on top of any other extension of sequential Bicolano. For our development, we only use the vertical extension framework. We give a brief sketch of this framework; for more information we refer to [8].

Figure 3 defines the signature of a vertical extension for the small step operational semantics. The type variable `state_t` defines the new state, whereas `bottom_state_t` refers to the type of the original Bicolano state. Every extension should preserve the original Bicolano state, *i.e.*, there should be a way to reconstruct the bottom state from the extension state (by defining `get_bottom_state`). Next, the extension defines the `step` relation between the new, extended states. Finally, one must define an `additional_step` relation that determines how to modify current state values to contain previously unforeseen bottom states at this level. Given an extended state s, and a bottom state b, it specifies an extended state s' with the additional information derived from s, such that the bottom state of s' is b. The purpose of this relation is to allow future extensions to reconstruct the extended state, on the basis of a bottom state only. The `add_step_step_compat` axiom specifies that `additional_step` should be compatible with the (extended) step relation.

In our formalization, we use the operations from the vertical extension framework described above. However, for a good understanding of our formal-

ization, it is sufficient to think about it as if we built the extension directly on top of the original Bicolano specification.

5 BicolanoMT

This section describes BicolanoMT[6], an interleaving semantics for multi-threaded Java bytecode, built on top of the existing Bicolano semantics. Based on the *Data Race Freeness* guarantee provided by the Java memory model (JMM) [17], we can consider an interleaving semantics only, abstracting away from all the details of the JMM. The guarantee ensures that all *correctly synchronized* programs only exhibit behaviors described by an interleaving semantics. Program analyses and logics proved with this semantics will thus be valid for correctly synchronized programs. We resort to existing data race detection static analyses to reject incorrectly synchronized programs [9,19,20]. Interestingly, programs containing data races can also be treated with our formalization, provided that the semantics of these races is described by some interleaving of the threads (*i.e.*, *benign data races*).

5.1 Data Structures

As previously explained, threads only communicate via the single shared memory (*i.e.*, the heap). All other thread information is local to a thread. Therefore, in our extension we keep a single heap, plus execution information per thread, by means of a partial map from thread IDs to thread execution information. Each thread contains a current (possibly exceptional) frame and a callstack, as in the Bicolano state, plus some additional information needed to model synchronization and interruption: the thread state, and the interruption state, as discussed below.

```
Module Type THREAD.
 Inductive t : Type :=
   normal : Frame.t -> CallStack.t -> ThreadState.t -> Interruption -> t
 | exception : ExceptionFrame.t -> CallStack.t -> ThreadState.t ->
               Interruption -> t.
End THREAD.
```

The definition of thread state describes under which conditions a thread can execute, *cf.* Lea [14, Ch. 1]. If a thread is in `runnable` state it is able to take a step in the semantics. When a thread has no more instructions to execute, its state becomes `terminated`. A thread is in state `blocked` when it unsuccessfully tried to acquire a lock, represented by the `Location` value; a thread can only leave the `blocked` state if it acquires the lock, or if it is interrupted by another thread. The last state is `waiting`, which is both used to represent the state where a thread issued a `wait` and it has not been notified, and where a thread issued a join on a thread that is not yet terminated. We

[6] The complete formalization can be foud at:
http://mobius.inria.fr/twiki/bin/view/BicolanoMT/

```
Module Type THREADMAP.
 Parameter t : Set.
 Parameter get : t -> ThreadId.t-> option Thread.t.
 Parameter update : t -> ThreadId.t -> Thread.t -> t.
 Parameter notifyAll : t -> Location -> ThreadSet.t -> t.
 Axiom get_notifyAll_in : forall tmap loc tset tid fr sf tinfo, ThreadSet.
  In tid tset = true -> get tmap tid = Some (Tr fr sf tinfo) ->
   get (notifyAll tmap loc tset) tid = Some (Tr fr sf (blocked loc)).
 Axiom get_notifyAll_not_in : forall tmap loc tset tid, ThreadSet.
  In tid tset = false -> get tmap tid = get (notifyAll tmap loc tset) tid.
End THREADMAP.
```

Fig. 4. `ThreadMap` definition

use the notation `wait_lock` and `wait_join` to refer to these two different cases in the formalization. As in the case of `blocked`, this constructor is parameterized by the lock or thread that is being waited for (in a disjoint union type parameter).

```
Module Type THREADSTATE.
 Inductive t : Set := | runnable | terminated |
   blocked : Location -> t | waiting : (ThreadId.t+Location) -> t.
End THREADSTATE.
```

The `Interruption` state is an enumerated data type containing only two constructors: `non_interrupted` and `interrupted`, with the obvious meaning.

The execution states for the different threads are combined into a single `ThreadMap`, mapping thread IDs to thread local information, as shown in Figure 4. The thread map is defined by operations `get` and `update`, with obvious meaning, and `notifyAll`. The latter allows to modify the thread state of all the threads in a `ThreadSet` from `waiting` to `blocked`. The behavior of these operations is specified by several axioms.

The heap is extended to contain synchronization information by adding operations for locking (`lock` and `unlock`) and querying information about the monitor state. For modularity, we do this by defining a module type `HEAP_MT`, a fragment of which is presented in Figure 5. This extends the sequential Bicolano heap with several new operations.

The `lock` and `unlock` operations take the following arguments: a heap (`Heap.t`) that represents the heap to be updated; a reference (`Location`) corresponding to the object whose monitor should be (un)locked; and a thread ID (`ThreadId.t`) of the thread issuing the operation. Some extra operations are defined to correctly specify the behavior of the synchronization mechanism; for example, the `getLockLevel` operation returns the number of times a lock has been acquired by a certain thread.

The wait/notify mechanism as described in the JVM Specification specifies that every object is associated to a wait set where threads waiting for a certain monitor are accumulated; this bookkeeping is specified in our formalization by the `getWaitset` operation, that records thread identifiers of threads waiting for a monitor. The relation `waitFor_and_unlock` relates two heaps, where

```
Module Type HEAP_MT.
 Declare Module Heap : HEAP.
 Import Heap.
 Parameter lock : t -> Location -> ThreadId.t -> t.
 Parameter unlock : t -> Location -> ThreadId.t -> t.
 Parameter getLockLevel : t -> Location -> ThreadId.t -> nat.
 Parameter getWaitLockLevel : t -> ThreadId.t -> nat.
 Parameter getWaitset : t -> Location -> ThreadSet.t.
 Inductive waitFor_and_unlock h loc tid h' : Prop :=
  waitFor_unlock_def : forall h'',
  unlockn h loc tid (getLockLevel h loc tid) h'' -> (* Release locks *)
  ... -> (* nothing changes ... *)
  (* except for getWaitLockLevel in the tid location ... *)
  getWaitLockLevel h' tid = getLockLevel h'' loc tid ->
  (* and the getWaitset in the loc location *)
  getWaitset h' loc = ThreadSet.add (getWaitset h'' loc) tid ->
  waitFor_and_unlock h loc tid h'.
End HEAP_MT.
```

Fig. 5. Fragment of Multi-threaded Heap definition

the latter is the result of releasing all the locks held in the former state by a certain thread, and adding this thread ID to the corresponding wait set. Several similar relations are defined (not shown here), and appropriate axioms are given, describing all the possible interactions between these operations.

Finally, Figure 6 defines multi-state to collect all information needed to describe the multi-threaded semantics . A multi-state contains a single *current* sequential support state, abstracted from the underlying semantics (`Support.state_t`), a thread map (`ThreadMap.t`) and the thread ID corresponding to that support state in the thread map. The idea is that only the thread with this ID can perform an action. One can think of the current thread as the thread being currently scheduled. Rescheduling of threads is modeled by the `update` operation, that allows to change the current support state. The `support_tid_coincidence` axiom requires that the correspondence between the current support state and the current thread identifier must be preserved, *i.e.*, the information contained in the thread map must be coherent with the information contained in the current support state for the current thread ID. The only heap present in the multi-state is the one contained in the current support state.

5.2 Operational semantics

Because of the use of the vertical extension framework we sometimes have to use three different semantics to describe how a step modifies the multi-state: the bottom semantics (*i.e.*, Bicolano as in [21]), the support semantics (as defined in [8]) of which we assume as little as possible, and finally, the multi-threaded semantics (and its corresponding state) as we are defining it. This might seem confusing at first, but in general the complexity added is not

```
Module Type MT_STATE.
 Parameter t : Type.
 Parameter get_current_support : t -> Support.state_t.
 Parameter get_current_tid : t -> ThreadId.t.
 Parameter get_tmap : t -> ThreadMap.t.
 Parameter update : t -> Support.state_t ->
                    ThreadId.t-> ThreadMap.t -> option t.
 Axiom support_tid_coincidence : forall st bst tr,
   Support.get_bottom_state (get_current_support st) bst ->
   ThreadMap.get (get_tmap st) (get_current_tid st) = Some tr ->
   bget_frame bst = trget_frame tr /\ bget_cstack bst = trget_cstack tr.
End MT_STATE.
```

Fig. 6. Fragment of Multi-state definition

significant, and it allows BicolanoMT to be used on top of previous extensions to Bicolano.

This paper describes only the most important cases of the semantics that is added as part of our extension; the Coq formalization contains all cases.

In the JVM, many of the mechanisms related to multi-threading are implemented as native methods (`start`, `notify` *etc.*). Therefore, they do not have a corresponding bytecode instruction, instead they are called using the `InvokeSpecial` instruction. Currently, the `InvokeSpecial` is not formalized in Bicolano. Therefore, we added bytecode place holders to model calls to these methods, and to formalize their semantics. The place holders have the name of the method, preceded by the keyword `_native_` (*e.g.*, `_native_start`).

Interleaving

First we consider interleaving of sequential instructions. Any thread that is enabled (*i.e.*, a `runnable` thread) can execute a step of the support semantics. The definition of the interleaving step determines the resulting state in the support semantics, and it updates the initial multi-state with the relevant information from this support state. Note that the only part of the support semantics whose existence we can assume is the bottom state; we extract the frame and callstack from it to update the threadmap accordingly.

Figure 7 shows the formalization of this state transformation. The multi-threaded semantics is represented in Coq by the inductive `mt_step` relation, that relates two states via the execution of a single instruction. Figure 7 shows only the interleaving of a normal step in the support semantics. The first line looks up, via the `current_info` definition, information from the multi-state (`mtst`) concerning the current support state (`sst`) the current thread ID (`tid`) and the thread map (`tmap`). The second line looks up information about the current thread; it checks that the thread is in `runnable` state, meaning that the thread is able to execute, and it binds the frame (`f`) and the callstack (`cs`). Next, the `no_synch_instr` predicate checks that the current instruction does not involve synchronization (a different case of the `mt_step` predicate covers those instructions). Then, `step` in the underlying semantics is consulted,

```
Inductive mt_step : Program -> state_t -> state_t -> Prop :=
| interleaved_normal_mtstep :
   forall p mtst mtst' sst sst' tid tmap tmap' h' cs cs' f f' it,
    current_info mtst sst tid tmap ->
    ThreadMap.get tmap tid = Some (Tr f cs runnable it) ->
    no_synch_instr f ->
    Support.step p sst sst' -> (* STEP IN SUPPORT *)
    Support.get_bottom_state sst' (St h' f' cs') ->
    tmap' = ThreadMap.update tmap tid (Tr f' cs' runnable it) ->
    Some mtst' = update mtst sst' tid tmap' -> (* UPDATE MULTISTATE*)
    mt_step p mtst mtst'
```

Fig. 7. Interleaving step

binding the result to the `sst'` variable. By using the mapping to the bottom semantics (`get_bottom_state`), the heap, current frame and callstack are extracted. Finally, the thread map and multi-state are updated.

Start

Start is a native method that causes a thread to be created and begin executing. A call to `start` on a thread object executes its `run` method. In our formalization, a new thread is created with the location of the thread object as thread ID. This is a formalization decision; the real JVM thread identifiers need not correspond to the formalization's `ThreadId` type.

Figure 8 shows two interesting details of the step start. First, two updates are required on the thread map; the first to modify the frame of the calling thread; and the second to add the newly created thread. Second, before updating the multi-state, the support state must be modified, to change the program counter and frame of the calling thread. However, since the `_native_start` instruction is not present in the underlying semantics, there is no step in the support semantics that does this. Instead, the new bottom state is constructed explicitly, and the `additional_step` relation of the support semantics is used to construct the new support state based on the knowledge of the old support state and the new bottom state. The rest of the formalization is similar to the interleaving case.

Monitorenter/Monitorexit

The `monitorenter` and `monitorexit` bytecode intructions lock and unlock Java monitors. Both take as parameter a reference whose lock must be acquired or released, respectively. Figure 9 shows the formalization of a `monitorenter` instruction that succeeds to acquire the lock[7]. First, the heap is checked for the state of the lock. If it is free or acquired by the same thread, `lockable` holds. In that case the lock is acquired by the thread, and the heap and state are updated. The case in which the lock is not free is not shown, but the main difference is that the heap is not updated, and the

[7] Only the relevant conditions are shown.

```
| native_start_mtstep_ok : ...
   current_info mtst sst tid tmap ->
   TMap.get tmap tid = Some (Tr f cs runnable it) ->
   f = (Fr m pc ((Ref loc)::s) l) ->
   instructionAt m pc = Some _native_start -> (* start place holder *)
   next m pc = Some pc' ->
   Heap.typeof h loc = Some (Heap.LocationObject cn) ->
   lookup p cn RunMethodSignature (cn, m') ->
   METHOD.body m' = Some bm' ->
   tmap' = TMap.update tmap tid (Tr (Fr m pc' s l) cs runnable it) ->
   new = Tr (Fr m' (BCMETHOD.firstAddress bm') OperandStack.empty
    (stack2localvar ((Ref loc)::s)  1)) nil runnable non_interrupted ->
   tmap'' = TMap.update tmap' loc new -> (* loc as tid (unique id) *)
   Support.additional_step p sst (St h (Fr m pc' s l) cs) sst' ->
   Some mtst' = update mtst sst' tid tmap'' ->
   mt_step p mtst mtst'
```

Fig. 8. Step `_native_start`

```
| monitorenter_nonblocking_mtstep_ok : ...
   instructionAt m pc = Some monitorenter -> ...
   lockable h loc tid ->
   h' = Heap_mt.lock h loc tid -> ...
   Support.additional_step p sst (St h' f' cs) sst' ->
   tmap' = ThreadMap.update tmap tid (Tr f' cs runnable it) ->
   Some mtst' = update mtst sst' tid tmap' ->
   mt_step p mtst mtst'
```

Fig. 9. Step `monitorenter`

thread state of the thread is changed to `blocked`, meaning that the thread is only able to re-attempt to acquire the lock. BicolanoMT also specifies all the exceptional cases documented in the JVM Specification. The formalization of `monitorexit` is similar, but the predicate `locked_by` is checked instead, and an `IllegalMonitorStateException` is thrown when the lock is not held.

Synchronized Methods

Synchronized methods constitute a special case of locking, where the object being locked is the one on which the method is invoked; in the case of `static` methods the lock is that of the `class` object allocated in the heap when the class is first loaded. The semantics of entering a synchronized method is very similar to that of `monitorenter`, but the way the lock object is looked up is different. This can be seen in Figure 10, where it is defined by the `is_invoke_synch` predicate, that in case the method is synchronized returns the reference to be locked. Also of interest is the exit of synchronized methods, where the lock must be released. Similar to `monitorexit`, the heap is consulted to check whether the thread holds the lock, and if not, an `IllegalThreadStateException` is thrown. To know whether a lock must be released, every method is checked for its synchronization modifier when it terminates, either because of a return or because of the propagation of an

```
| invoke_synch_ok : ...
   is_invoke_synch p sst loc ->
   Support.step p st sst' ->  ...
   lockable h' loc tid ->
   h'' = Heap_mt.lock h' loc tid ->
   Support.additional_step p sst' (St h'' f' cs') sst'' -> ...
   tmap' = ThreadMap.update tmap tid (Tr f' cs' runnable it) ->
   Some mtst' = update mtst sst'' tid tmap' -> (* Update the Tmap *)
   mt_mtstep p mtst mtst'
```

Fig. 10. Synchronized method invocation multi-step

```
| native_wait_enter_mtstep_ok :
   ThreadMap.get tmap tid =
                       Some (Tr f cs runnable non_interrupted) -> ...
   instructionAt m pc = Some _native_wait -> ...
   locked h loc tid -> Heap_mt.waitFor_and_unlock h loc tid h' -> ...
   f' = Fr m pc' s l ->
   Support.additional_step p sst (St h' f' cs) sst' ->
   tmap' = ThreadMap.update tmap tid (Tr f' cs (wait_lock loc)
           non_interrupted) ->
   Some mtst' = update mtst sst' tid tmap' ->
   mt_step p mtst mtst'
```

Fig. 11. Step `_native_wait`

uncaught exception.

Wait/Notify

When the `wait` method is invoked on an object, the caller releases all the locks held on this object. Then it blocks, entering the waiting state and joining the wait set for that lock. Figure 11 presents the case where a thread succeeds in entering the waiting state. A thread can leave the waiting state if either another thread calls `notify` and this thread is selected, or the `notifyAll` is called (both on the appropriate lock). When a thread is notified, its thread state becomes `blocked`, meaning that the thread must reacquire all the locks it released when waiting. Another way for a method to exit the `wait_lock` state is to be interrupted by another thread. Moreover, if a call to `wait` occurs in an interrupted thread (*i.e.*, its interruption state is `interrupted`), an exception is thrown and the thread continues to be runnable (see below).

The formalization of `notify` and `notifyAll` is similar to `wait`, except that a (resp. all) thread(s) in the waitset is removed, and its state changes from `waiting` to `blocked` – to proceed, the notified thread must re-acquire the locks that it released when waiting. The conditions for `notify` and `notifyAll` to succeed are the same as for wait: the lock must be held by the calling thread (otherwise `IllegalMonitorStateException` is thrown), and the object must not be null (otherwise a `NullPointerException` is thrown).

```
| native_wait_mtstep_InterruptedException :
   ThreadMap.get tmap tid = Some (Tr f cs runnable interrupted) -> ...
   instructionAt m pc = Some _native_wait -> ...
   locked h loc tid ->
   Heap.new h p (Heap.LocationObject
                 (javaLang,InterruptedException)) = Some (loc, h') ->
   f' = FrE m pc loc l ->
   tmap' = TM_update tmap tid (TrE f' cs runnable non_interrupted) ->
   Support.additional_step p sst (StE h' f' cs) sst' ->
   tmap' = ThreadMap.update tmap tid sst' ->
   Some mtst' = update mtst sst' tid tmap' ->
   mt_step p mtst mtst'
```

Fig. 12. Interrupted wait exit step

Interruption

The last of the mechanisms we will describe is interruption. A thread can be interrupted at any moment; its actual effect depends on the particular thread state of the interrupted thread. If the state is waiting or joining (*i.e.*, waiting for a thread to terminate), the `interrupt` method causes the thread to become runnable and throw an interrupted exception. In any other case, it causes the thread to be marked as `interrupted`. This mark (flag) can be reset at any moment by the `interrupted` method. If an interrupted thread executes a `wait` or a `join` method it causes the interruption state to be reset and to throw an `InterruptedException`, as well as reseting the thread state to `runnable`.

Figure 12 shows the case where a `wait` method is called, while the interruption state value is `interrupted`. As described, an `InterruptedException` is allocated on the heap, and a reference (`Location`) to the exception is stored in the current frame of the calling thread. Then the thread state is reset to `runnable` and `non_interrupted`, and an exceptional thread state is stored in the thread map. The other cases are specified similarly.

Concluding remarks

It is important to notice that all the exceptional cases, as well as all possible interactions of the different thread states and instructions (as well as native methods) are specified in the semantics. Therefore the number of cases considered is very large (around 500 lines of Coq specification). However, we do not formalize thread groups, timing and class loading, as these are not allowed in the MIDP framework.

6 Related Work

Moore and Porter formalize a significant part of the Java bytecode instruction set in the ACL2 theorem prover [18]. Although their semantics includes multi-threading aspects, it does so in a simple and minimalistic way, that is sufficient for them to prove properties of several example programs. Their

formalization contains the `monitorenter` and `monitorexit` synchronization instructions and thread creation, but synchronized methods, the wait/notify mechanism and interruptions are not taken into account. Moreover, their formalization does not consider exceptions, which are an important source of complexity in our formalization.

Belblidia and Debbai [4] present an operational semantics for Java bytecode that supports multi-threading. Their semantics covers many concurrency-related aspects: synchronization via `monitorenter` and `monitorexit`, as well as synchronized methods; thread creation; and thread termination. However, most native methods (wait/notify, interruption, join) related to concurrency are not handled in their semantics. Further, we completely separate the single threaded semantics from the multi-threaded one, *i.e.*, no multi-threading related bytecode instructions or knowledge is present in Bicolano, while Belblidia and Debbai specify instructions related to multi-threading in their single-threaded semantics (the first semantic layer in their work). Their single-threaded semantics generates labels to signal the multi-threaded layer how to react; thus they do not have a clear distinction between sequential and multi-threaded semantics. Finally, in contrast to our work, they do not have a tool-supported formalization.

Lochbihler gives a source code level formalization of multi-threaded Java in the Isabelle theorem prover [16]. This models synchronization as well as the wait/notify mechanism, but it does not contain interrupts or join (it is only remarked that this could easily be added). Since the semantics is given for Java source code, it is at some points more artificial than ours. We consider that bytecode is the most appropriate level to specify an interleaving semantics for multi-threading, as it has the right level of granularity. Moreover, a source code level formalization cannot model unstructured monitor acquisition, that can be produced by compiler optimizations. Further, Lochbihler considers synchronized methods to be syntactic sugar for `monitorenter` and `monitorexit`. This is not appropriate for bytecode, where monitors might be acquired and released in different order. Also here, the single threaded semantics produces labels for the higher level interleaving semantics. This sometimes results in rather awkward sequences of instructions, like releasing a lock and reacquiring it immediately, to check whether a thread holds a lock.

Finally, Stärk, Schmid and Börger define an abstract state machine semantics for multi-threaded Java [22] covering most of the synchronization mechanisms presented here. However, their formalization is for source code and they do not formalize it in a theorem prover.

7 Conclusions

We have formalized BicolanoMT in Coq. To the best of our knowledge, this is the most complete interleaving semantics for multi-threaded Java. It features thread creation and termination; monitor synchronization; the wait/notify

mechanism; interruption; synchronized methods and synchronization-aware exceptions and method termination. BicolanoMT is modular in the sense that it only models the multi-threading aspects of Java, building on top of the Bicolano semantics for sequential Java, using the extensional framework of Czarnik and Schubert [8]. BicolanoMT can be used to prove soundness of concurrent extensions of sequential type systems proved sound in Bicolano. For example, the information flow type system developed and formalized for the sequential case in [2], has been extended to concurrency in [3]. We can reuse the proof of soundness of the former by adding the new multi-threaded cases, and lifting the definitions of the old cases by means of the `get_bottom_state` relation. Data race detection type systems (as in [9]) can be proved correct using our semantics. Also, we plan to make a connection with our formalization of the Java Memory Model [11], and prove that they are isomorphic for data race free programs.

Acknowledgements

We thank David Pichardie, Aleksy Schubert and Patryk Czarnik for comments on and help with the formalization. We are especially grateful to Allard Kakebeen for his contributions to the formalization. And finally we would like to thank the reviewers for their kind comments and remarks.

References

[1] G. Barthe and G. Dufay. A Tool-Assisted Framework for Certified Bytecode Verification. In M. Wermelinger and T. Margaria-Steffen, editors, *Proceedings of FASE'04*, volume 2984 of *LNCS*, pages 99–113, Barcelona, Spain, March 2004. Springer.

[2] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. In R. De Niccola, editor, *European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 125 – 140. Springer-Verlag, 2007.

[3] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *European Symposium On Research In Computer Security*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[4] N. Belblidia and M. Debbabi. A dynamic operational semantics for JVML. *Journal of Object Technology*, 6:71–100, 2007.

[5] Mobius Consortium. Deliverable 3.1: Bytecode specification language and program logic. Available online from `http://mobius.inria.fr`, 2006.

[6] Mobius Consortium. Deliverable 4.3: Intermediate report on proof-transforming compiler. Available online from `http://mobius.inria.fr`, 2007.

[7] Coq development team. The Coq proof assistant reference manual V8.0. Technical Report 255, INRIA, France, March 2004. http://coq.inria.fr/doc/main.html.

[8] P. Czarnik and A. Schubert. Extending operational semantics of the Java bytecode. In *Trustworthy Global Computing*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To appear.

[9] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Programming Languages Design and Implementation*, pages 219–232, New York, NY, USA, 2000. ACM Press.

[10] N.G. Fruja. *Type Safety of C# and .NET CLR*. PhD thesis, ETH Zurich, 2006.

[11] M. Huisman and G. Petri. The Java memory model: a formal explanation. In *VAMP 2007: Proceedings of the 1st International Workshop on Verification and Analysis of Multi-threaded Java-like Programs*, 2007. Technical Report ICIS-R07021, Radboud University Nijmegen.

[12] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[13] T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1998.

[14] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns (Second Edition)*. Addison-Wesley, Boston, MA, USA, 1999.

[15] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification. Second Edition*. Sun Microsystems, Inc., 1999. http://java.sun.com/docs/books/vmspec/.

[16] A. Lochbihler. Type safe nondeterminism - a formal semantics of Java threads. In *ACM Workshop on Foundations of Object-Oriented Languages*, 2008.

[17] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Principles of Programming Languages*, pages 378–391, 2005.

[18] J. Strother Moore and G. Porter. The apprentice challenge. *ACM Transactions on Programming Languages and Systems*, 24(3):193–216, 2002.

[19] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Principles of Programming Languages*, pages 327–338, New York, NY, USA, 2007. Association of Computing Machinery.

[20] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Programming Languages Design and Implementation*, pages 308–319, 2006.

[21] D. Pichardie. Bicolano – Byte Code Language in Coq. http://mobius.inria.fr/bicolano. Summary appears in [5], 2006.

[22] R. F. Stärk and J. Schmid and E. Börger. *Java and the Java Virtual Machine—Definition, Verification, Validation*. Springer-Verlag, 2001.

[23] Sun Microsystems Inc., 4150 Network Circle, Santa Clara, California 95054. *Connected Limited Device Configuration.Specification Version 1.1. Java™ 2 Platform, Micro Edition (J2ME™)*, March 2003.