JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 6, No. 3, March–Aprile 2007

# A Dynamic Operational Semantics for JVML

**Nadia Belblidia**
Computer Security Laboratory, Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada.
**Mourad Debbabi**
Computer Security Laboratory, Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada.

In this paper[a], we present a dynamic semantics for the Java Virtual Machine Language (JVML). The semantics is specified in an operational style according to the syntactic structure of JVML programs. In order to ascribe meanings to threading, the semantics is made small-step and is structured in two layers: The first layer consists of judgements that capture the semantics of sequential JVML programs in isolation. The second layer consists of judgements that capture the parallel execution of JVML threads. The semantics presented in this paper is a faithful and formal transcription of JVML specification as described in [1]. Besides, we provide full account details for the most technical and tricky aspects of JVML such as multi-threading, synchronization, method invocations, exception handling, object creation, object's fields manipulation, stack manipulation, local variable access, modifiers, etc. The presented semantics is, to the best of our knowledge, the first dynamic semantics of JVML that provides semantics for that many features within the same framework.

[a]This research is funded under the NSERC DND partnership program in collaboration with DRDC Valcartier(Defence Research and Development Canada) and Bell Canada.

## 1 MOTIVATIONS AND BACKGROUND

Java is a very popular and appealing language to millions of software developers. Moreover, it is largely deployed in the enterprize world since it powers a plethora of applications and services. Its market share is rapidly increasing and nowadays it is heavily present in the most prominent e-business platforms. Furthermore, Java is omnipresent in mobile Internet-enabled devices such as cellular phones (more than 750 millions of Java-enabled units are already deployed). It is then of paramount importance to measure and control the key quality attributes of the Java platform. An essential step towards such an aim is to grasp the semantic underpinnings of the Java platform, which is very well known to be subtle, complex, sophisticated and highly technical. Therefore, there is a desideratum in compiling this semantics into a structured, rigorous, robust, and faithful description.

The primary objective of this work is to grasp the semantics of Java runtime (Java Virtual Machine Language or JVML) and to compile the underlying meanings into a formal dynamic semantics. JVML is the language interpreted by the Java Virtual Machine (JVM), which is the heart of any Java platform.

One can ask a legitimate question: Why another semantics for JVML? The right answer to this question is:

- In spite of the intensive activities of the research community in formalizing JVML semantics, it remains that there is no contribution that formally addresses, within the same framework, the meanings of JVML features such as multi-threading, synchronization, exception handling, the four method invocations and the use of modifiers.
- Most of the proposed research contributions so far consider only one single thread of execution even though multi-threading is a keystone in Java.
- In the very few proposals where multi-threading has been addressed, it has been done in a way that is no faithful to the official JVML specification. For instance, no distinction is made between implementing the interface `Runnable` or extending the class `Thread`.

Besides these arguments, the main motivation that led us to the formalization of JVML stems from a security investigation of the Java platform. Actually, we needed a formalization that accounts, at the same time and within the same framework, for all the aforementioned JVML aspects. Such a requirement was not satisfied by the state of the art contributions on JVML semantics. The semantics that we report in this paper is a small-step operational style structured in two-layers dynamic semantics: The first layer consists of judgements that capture the semantics of sequential JVML programs in isolation. The second layer consists of judgements that capture the parallel execution of JVML threads.

The rest of the paper is organized as follows. Section 2 is a discussion of the state of the art on Java and JVML semantics. Section 3 is devoted to the description of the JVML syntax. Section 4 describes the ingredients that are used in the semantic description: Data types, computable values, dynamic environments, stores, frames and configurations. The semantic rules are given in section 5 and finally some concluding remarks are reported in section 6.

## 2 RELATED WORK

The closest proposals to our contribution are those defining a formal semantics to JVML. Most of the research initiatives describing the semantics of JVML subsets use a small-step structural operational semantics [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]. The main objective of these proposals is either to purely study the semantics or put the emphasis on typing constraints. In his papers [2, 3], Bertelsen presents a very detailed semantics of a subset of JVML. However, his work does not address the semantics of multi-threading and synchronization (no meaning is ascribed to `monitorenter` and `monitorexit` instructions). Freund and Mitchell, in [6], use a type system to investigate the problem of object initialization and subroutines. The authors add support for objects, classes, interfaces, arrays and exceptions in [5, 7]. Nevertheless, Freund and Mitchell's contributions considers a mono-threaded virtual machine (no support for multi-threading and synchronization). This is understandable since the primary intent of the authors was to put the emphasis on initialization rather than completeness of JVML semantics. Hagiya and Tozawa in [8] and Klein and Wildmoser in [9] define operational semantics for JVML subsets with subroutines in order to explore type safety issues. However, in these proposals, method invocations, exception handling and multi-threading have not been taken into account. In [4], Bigliardi and Laneve isolate a sub-language of JVML with thread creation and mutual exclusion and define an operational semantics together with a formal verifier that enforces basic safety properties on threads involving lock and unlock operations. In their work, the authors did not handle modifiers and did not consider the instructions `invokespecial`, `invokestatic` and `invokeinterface`. In addition, their semantic handling of `invokevirtual` is not compliant with the official specification since they only consider void methods without arguments. Furthermore, they did not address an interesting and subtle issue, which is the semantic treatment of thread creation by either extending the class `Thread` or by implementing the interface `Runnable`. In fact, they use an instruction `start`($\sigma$) as an artifact to the instruction `invokevirtual java/lang/Thread/start()`. In another similar paper of Laneve [10], the invoke method instructions have not been taken into account. Siveroni [11, 12] presents an operational semantics for a language that models the Java Card virtual machine including exception handling, array objects and subroutines but lacking support for multi-threading aspect. Stata and Abadi [13] use type systems for bytecode verification focussing on subroutine safety. They define an operational semantics for a JVML subset containing 9 instructions `inc, pop, push, load, store, if, jsr, ret and halt`. Again this is highly understandable since the authors were mainly interested in subroutine safety. Börger and Schulte [14] use operational semantics of Abstract State Machines (ASMs) to describe the JVM with the goal of defining a platform for correct compilation of Java code.

None of the previously proposed semantics of JVML handles, in a thorough way, the semantics of all the major aspects of the language, at the same time and within the same framework.

## 3 JVML SYNTAX

In this section, we present the syntax of JVML. It is presented in Table 1 in the BNF form together with the notation that is used along this paper.

| | | | | |
|---|---|---|---|---|
| *Instruction* | ::= | `aload` $i$ | \| | `iload` $i$ |
| | \| | `astore` $i$ | \| | `istore` $i$ |
| | \| | `pop` | \| | `push` $n$ |
| | \| | `dup` | \| | `iadd` |
| | \| | `new` $i$ | \| | `ifeq` *adr* |
| | \| | `ifne` *adr* | \| | `goto` *adr* |
| | \| | `return` | \| | `ireturn` |
| | \| | `areturn` | \| | `athrow` $i$ |
| | \| | `monitorenter` | \| | `monitorexit` |
| | \| | `invokevirtual` $i$ | \| | `invokespecial` $i$ |
| | \| | `invokestatic` $i$ | \| | `invokeinterface` $i$,$n$ |
| | \| | `getstatic` $i$ | \| | `putstatic` $i$ |
| | \| | `getfield` $i$ | \| | `putfield` $i$ |

Table 1: JVML Bytecode Grammar

## Notation

- Given a map $m$ from $A$ to $B$, the domain of $m$, $A$, is written $\text{Dom}(m)$.
- Given a map $f$, we write $f[x \mapsto v]$ to denote the updating operation of $f$ that yields a map that is equivalent to $f$ except that $x$ is from now on associated with $v$.
- Given a record space $D = \langle f_1 : D_1, f_2 : D_2, \ldots, f_n : D_n \rangle$ and an element $e$ of type $D$, the access to the field $f_i$ of $e$ is written $e.f_i$ and the update of the fields $f_{i1}, \ldots, f_{ik}$ in $e$ by the values $v_{i1}, \ldots, v_{ik} \in D_{i1}, \ldots, D_{ik}$ is written $e[f_{i1} \leftarrow v_{i1}, \ldots, f_{ik} \leftarrow v_{ik}]$. If an update of a field $f_{ij}$ with a value $v_{ij}$ is subjected to a condition $C$, we will use the notation $e.[\ldots, f_{ij} \leftarrow v_{ij}/C, \ldots]$.
- Given a type $\tau$, we write `(`$\tau$`)-list` to denote the type of lists having elements of type $\tau$.
- Given a type $\tau$, we write `(`$\tau$`)-set` to denote the type of sets having elements of type $\tau$.
- The space `Identifier` classifies identifiers whereas *NoneType* classifies the unique value `None`.

## 4 SEMANTIC INGREDIENTS

In this section, we define the ingredients that are used in the semantic description. Accordingly, we introduce and define the notions of data type, computable value, environment, memory store, frame and configuration. A data type refers to a type

| | | | | |
|---|---|---|---|---|
| *Type* | ::= | *PrimitiveType* | \| | *ReferenceType* |
| *ReferenceType* | ::= | *ClassType* | \| | *InterfaceType* |
| *ResultType* | ::= | *Type* | \| | `void` |
| *ClassType* | ::= | `Identifier` | | |
| *InterfaceType* | ::= | `Identifier` | | |

Table 2: Type Algebra

| | | | | | | |
|---|---|---|---|---|---|---|
| *Value* | ::= | *Location* | \| | *Constant* | \| | `Null` |

Table 3: Runtime Values

that is used in JVML. A computable value refers to a dynamic value that is the result of a semantic evaluation of a given JVML expression. An environment is the context that holds the definitions under which the evaluation is done. It corresponds to the actual constant pool of a class file. Memory store is an abstraction of both the memory storage and the heap. The proposed semantics has the form of a small step operational semantics that is based on evolving configurations.

## Type Algebra

We describe in Table 2 the type algebra. We consider two categories of types: Primitive types and reference types. Reference types are either class types or interface types. An object is a dynamically created class instance and reference values are pointers to these objects.

## Computable Values

We define in Table 3 the computable values. Two kinds of values are considered: Locations and constants. Locations are addresses and constants are values of primitive types. The particular reference value `Null` refers to no object.

## Environment

We define hereafter dynamic environments. We assume that the reader is familiar with the class file format as described in the official specification of JVML [1]. The environment as described in Table 4 and Table 5 models the different program declarations and is represented as a map that associates a set of classes to a set of reference types. A class is a record containing a constant pool, a super-class, a set of interfaces, a list of fields, a map that associates values to static fields, a list of methods, two flags that indicate whether the class is initialized or not and if the class is an interface and a monitor. A constant pool is a map that associates a set of integers with a set of constant pool entries. A constant pool entry can be a class type, a pair of a method signature and a supposed class, or a pair of a field signature and a supposed class. The supposed class represents the class in which the method or the field is supposed to be found. The value `None` for the super class indicates that the class does not have a super class. The monitor associated with a class is a record of three components: *threadOwner* (thread identifier that locked the class), *depth* (the number of times this class has been locked by this same thread) and a *waitList*(a list of all the threads blocked waiting for this class). A method consists of a method signature, a class name in which the method has been defined, a set of modifiers, a bytecode sequence, a list of method variables and an exception table. A method signature is a record that contains the method's name, the types of arguments and the result type. The list of the method variables contains the default values of all local variables defined inside the method. The method's parameters are not considered in the method variables.

An exception table is a list of exception handlers where an exception handler is defined by:

- Two natural numbers, *startPc* and *endPc*, that are used to determine the code range where the exception handler is valid.
- A natural number, *handler*, that indicates the location that is called upon exception.
- A class type, *exceptionType*, that indicates the class of the exception.

A field is represented by a record that contains a field signature, a class type to which the field belongs to and a set of modifiers. The signature is a combination of the field's name and the field's type.

## Memory Store

We define, in what follows, a model that captures both the memory storage and the heap of the Java virtual machine. The store as shown in Table 6 is a partial mapping from locations to Java objects that are class instances. A Java object is a record containing the class type of the object, a map from the object fields identifiers to computable values, a monitor and an additional information *fromRunnable*. If the object is a `Thread` instance constructed from a class that implements the interface `Runnable`, the name of this class is put in the field *fromRunnable* otherwise *fromRunnable* is set to the value `None`. This information is useful when a method `start` is invoked on an object that is an instance of `Thread` or one of its subclasses. It allows to know which method `run` to execute. The lookup of the `run` method starts from the dynamic class of the object if *fromRunnable* is `None` otherwise it starts from *fromRunnable*.

| | | | |
|---|---|---|---|
| *Environment* | ::= | *ReferenceType* $\xrightarrow[m]{}$ *Class* | |
| *Class* | ::= | ⟨ *constantPool:* | *ConstantPool,* |
| | | *superClass:* | *ClassType* \| *NoneType,* |
| | | *interfaces:* | (*ClassType*)`-set`, |
| | | *fields:* | (*Field*)`-list`, |
| | | *staticMap:* | *Field* $\xrightarrow[m]{}$ *Value,* |
| | | *methods:* | (*Method*)`-list`, |
| | | *initialized:* | *Int,* |
| | | *interface:* | *Int,* |
| | | *monitorClass:* | *Monitor* ⟩ |
| *ConstantPool* | ::= | *Int* $\xrightarrow[m]{}$ *ConstantPoolEntry* | |
| *ConstantPoolEntry* | ::= | *ClassType* | |
| | \| | ⟨ *methodSignature:* | *MethodSignature,* |
| | | *supposedClass:* | *ClassType* ⟩ |
| | \| | ⟨ *fieldSignature:* | *FieldSignature,* |
| | | *supposedClass:* | *ClassType* ⟩ |
| *Monitor* | ::= | ⟨ *threadOwner:* | *ThreadOwner,* |
| | | *depth:* | *Nat,* |
| | | *waitList:* | *WaitingList* ⟩ |
| *ThreadOwner* | ::= | *ThreadId* \| *NoneType* | |
| *WaitingList* | ::= | (*ThreadId*)`-list` | |
| *ThreadId* | ::= | *Nat* | |
| *Field* | ::= | ⟨ *fieldSignature:* | *FieldSignature,* |
| | | *fromClass:* | *ClassType,* |
| | | *fieldModifiers:* | (*FieldModifier*)`-set` ⟩ |
| *Method* | ::= | ⟨ *methodSignature:* | *MethodSignature,* |
| | | *fromClass:* | *ClassType,* |
| | | *methodModifiers:* | (*MethodModifier*)`-set`, |
| | | *code:* | *Code,* |
| | | *methodVariables:* | *MethodVariables,* |
| | | *exceptionTable:* | *ExceptionTable* ⟩ |
| *Name* | ::= | `Identifier` | |
| *Code* | ::= | *ProgramCounter* $\xrightarrow[m]{}$ *JVML Instruction* | |
| *ProgramCounter* | ::= | *Nat* | |
| *FieldSignature* | ::= | ⟨ *name:* | *Name,* |
| | | *type:* | *Type* ⟩ |

Table 4: Environment Part I

| | | | |
|---|---|---|---|
| *MethodSignature* | ::= | ⟨ *name:* | *Name*, |
| | | *argumentsType:* | (*Type*)-list, |
| | | *resulType:* | *Type* ⟩ |
| *MethodVariables* | ::= | (*Value*)-list | |
| *ExceptionTable* | ::= | (*ExceptionHandler*)-list | |
| *FieldModifier* | ::= | public \| private \| static | |
| *MethodModifier* | ::= | public \| private \| static \| synchronized | |
| *ExceptionHandler* | ::= | ⟨ *startPc:* | *Nat*, |
| | | *endPc:* | *Nat*, |
| | | *handler:* | *Nat*, |
| | | *exceptionType:* | *ClassType* ⟩ |

Table 5: Environment Part II

| | | | |
|---|---|---|---|
| *Store* | ::= | $Location \xrightarrow[m]{} JavaObject$ | |
| *JavaObject* | ::= | ⟨ *classType:* | *ClassType*, |
| | | *fieldsMap:* | $Field \xrightarrow[m]{} Value$, |
| | | *monitor:* | *Monitor* |
| | | *fromRunnable:* | *ClassType* \| *NoneType* ⟩ |
| | | *waitList:* | *WaitingList* |

Table 6: Store Structure

An object monitor has the same structure than a class monitor and is set to $\langle$None$, 0, [\ ]\rangle$ if the object is not locked.

## Frame

A frame is a runtime data structure that captures the execution state of a JVML method. It is defined as a tuple $\{\!|m, pc, l, o, z|\!\}$ where:

- $m$ refers to the current method.
- $pc$ denotes the program counter that refers to the address of the instruction to be executed in the method $m$.
- $l$ holds the values of the different local variables of $m$.
- $o$ holds the stack of operand values.
- $z$ holds the element locked when $m$ is a synchronized method. It is the class locked in the case of a static method or the reference to the locked object in

| | | | |
|---|---|---|---|
| *MethodFrame* | ::= | ⟨ *method* | *Method,* |
| | | *programCounter:* | *ProgramCounter* , |
| | | *locals:* | *Locals,* |
| | | *operandStack:* | *OperandStack,* |
| | | *synchronizedElement:* | *SynchronizedElement* ⟩ |
| *Locals* | ::= | (*Value*)`-list` | |
| *OperandStack* | ::= | (*Value*)`-list` | |
| *SynchronizedElement* | ::= | *ClassOrLocation* \| *NoneType* | |
| *ClassOrLocation* | ::= | *Location* \| *ClassType* | |

Table 7: Method Frame

the case of a non static method. The value is `None` in case of non synchronized methods.

The formal description of the frame is given in Table 7.

## Configurations

We need to introduce two categories of configurations. The domain of configurations that are dedicated to mono-threaded programs is *ThreadConfiguration*. The domain of configurations that are dedicated to multi-threaded programs is *MultiThreadConfiguration*. These two categories are respectively defined in Table 8 and Table 9. A thread configuration will have the following form:

$$\langle \mathcal{E}, \mathcal{S}, \{\!|m, pc, l, o, z|\!\} :: \mathcal{F}, \mathcal{L}, \iota, x\rangle$$

where:

- $\mathcal{E}$ represents the environment.
- $\mathcal{S}$ is the store.
- $\{\!|m, pc, l, o, z|\!\} :: \mathcal{F}$ represents the thread's stack from which method frames are retrieved. The term $\{\!|m, pc, l, o, z|\!\}$ denotes the frame that is the top element of frame stack.
- $\mathcal{L}$ contains the objects and classes that are locked by the current thread.
- $\iota$ represents the identity of the current thread i.e. the one executing the method $m$.
- $x$ indicates if an exception has been detected. Whenever an exception is raised, we will use the location to point to the underlying object that is an instance

| | | | |
|---|---|---|---|
| *ThreadConfiguration* | ::= | *Environment* × *Store* × *ThreadInformation* | |
| *ThreadInformation* | ::= | ⟨ *threadStack:* | *ThreadStack*, |
| | | *lockedElements:* | *LockedElements* |
| | | *threadId:* | *ThreadId* |
| | | *exception:* | *Exception* ⟩ |
| *ThreadStack* | ::= | (*MethodFrame*)`-list` | |
| *LockedElements* | ::= | (*ClassOrLocation*)`-list` | |
| *Exception* | ::= | *Location* \| *NoneType* | |

Table 8: Thread Configuration

| | | | |
|---|---|---|---|
| *MultiThreadConfiguration* | ::= | *Environment* × *Store* × *JavaStack* | |
| *JavaStack* | ::= | $\mathit{ThreadId} \underset{m}{\rightarrow} \mathit{Thread}$ | |
| *Thread* | ::= | ⟨*threadInformation:* | *ThreadInformation*, |
| | | *state:* | *State* ⟩ |
| *State* | ::= | `active` \| `blocked` | |

Table 9: Multi-Threads Configuration

of the class `Throwable`. If no exception is thrown `None` is used instead of a location $e$.

The configuration in Table 9 is used in presence of multiple threads and is a combination of an environment, a store and a Java stack. The Java stack contains information about the current threads and consists of a partial mapping that associates thread information and state to the *Nat* number that identifies the thread. The term `blocked` is used to denote the state of a thread waiting for a resource, owned actually by another thread, otherwise the thread is said to be `active`.

## 5 SEMANTICS RULES

The semantics is structured in two layers, one for threads in isolation and another for threads running in parallel following [15]. For processes in isolation, the semantics is defined using a labelled transition system on thread configurations i.e. $(\mathit{ThreadConfiguration}, \Lambda, \longrightarrow)$ whereas an unlabelled state transition system $(\mathit{MultiThreadConfiguration}, \hookrightarrow)$ is used for multisets of threads. The set of labels $\Lambda$ is defined as follows:

$$
\begin{array}{rcll}
\Lambda \ni \ell & ::= & \epsilon & \\
& | & \mathit{block} \ldots\ldots\ldots\ldots & \mathit{Block\ Current\ Thread} \\
& | & \mathit{kill} \ldots\ldots\ldots\ldots & \mathit{Kill\ Current\ Thread} \\
& | & \mathit{run}(\mathit{class} : \mathit{ClassType}) \ldots\ldots & \mathit{Fork\ New\ Thread} \\
& | & \mathit{notify}(x : \mathit{ClassOrLocation}) \ldots & \mathit{Notify\ Blocked\ Threads}
\end{array}
$$

The labels on the transitions contain the information to send from the first layer to the second one. The transition label $\epsilon$[1] allows to report, in the Java stack, all the modifications that the current thread has been subjected to during this transition. The transition label *block* is designed to instruct the second layer of the dynamic evaluation to change the current thread's state from `active` to `blocked`. The *kill* label refers to the case where the current thread must be killed and its corresponding entry in the Java stack removed. The label $\mathit{run}(\mathit{class}{:}\mathit{ClassType})$ reports to the second layer that a new thread must be created and that the lookup of its `run` method must start from the parameter *class*. The last transition label $\mathit{notify}(x{:}\mathit{ClassOrLocation})$ is used by the second layer in order to change the state of all the threads waiting for the resource $x$ from `blocked` to `active`.

## First Layer

Because of the imposed restriction on space, we cannot present the totality of the semantic rules. Therefore, we made the choice to report a significant part of the first layer rules. As of the rules of the second layer, they are completely reported.

We selected for each transition label a number of rules. The semantic rules of the first layer count only one rule with the transition label *run* and one rule with the transition label *kill*. We present first these two rules. Afterwards, we will present four different rules for each of the remaining labels i.e. $\epsilon$, *block* and *notify*.

The transition in the following rule is labelled with the *run* label. It represents the semantics of `invokevirtual` $i$ when the method invoked is the `start` method of the class `Thread` or one of its subclasses. In this case, an information is sent to the second level using the label $\mathit{run}(\mathit{class})$ in order to start running the new thread. The parameter *class* of the label represents the class from which the lookup for the method `run` of the new thread starts. If the object referenced by the top of the operand stack was constructed using a `Runnable` interface then the field *fromRunnable* of this object is assigned to the variable *class*. If the object referenced by the top of the operand stack was not constructed using a `Runnable` interface (ie. the field *fromRunnable* is `None`) then its dynamic class (ie. the class `Thread` or one of its subclasses from which the object has been instantiated) is assigned to the variable *class*. When the rule processes, the reference on the top of the operand stack of the current thread is popped, the current thread program counter is incremented.

[1]For the rest of the paper, we adopt the notation $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$ instead of $\mathcal{C}_1 \xrightarrow{\epsilon} \mathcal{C}_2$.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{invokevirtual}\ i \\ ms = \mathsf{thisConstantPoolEntry}(\mathcal{E}, m, i).methodSignature \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{E}, m, i).supposedClass \\ \mathsf{isMethResolved}(\mathcal{E}, ms, ct) \\ argCount = \mathsf{length}(ms.argumentsTypes) \\ Loc = \mathsf{getOneStackElem}(o, argCount) \\ Loc \neq \texttt{Null} \\ dc = \mathsf{getDynamicClass}(\mathcal{S}, Loc) \\ m' = \mathsf{lookupM}(\mathcal{E}, ms, dc) \\ m' \neq \texttt{None} \\ m'.name = \texttt{start} \wedge \mathsf{isThread}(m'.fromClass) \\ class = \mathsf{ifThenElse}(\mathcal{S}(Loc).fromRunnable = \texttt{None}, dc, \mathcal{S}(Loc).fromRunnable) \\ o' = \mathsf{popStack}(o, 1) \end{array}}{\begin{array}{c} \langle \mathcal{E}, \mathcal{S}, \{\!| m, pc, l, o, z |\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \xrightarrow{run(class)} \\ \langle \mathcal{E}, \mathcal{S}, \{\!| m, pc+1, l, o', z |\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \end{array}}$$

The next rule shows a transition that uses the *kill* label. It describes the case where the exception flag in the initial thread configuration is an exception reference $e$ and where the frames stack contains only one method. Furthermore, this unique method in the frames stack can not handle the exception referenced by $e$. In this case, the information that the current must be killed is sent to the second layer using the transition label *kill*.

$$\frac{\begin{array}{c} type = \mathsf{getDynamicClass}(\mathcal{S}, e) \\ \mathsf{appropriatePcHandler}(\mathcal{E}, pc, type, m.exceptionTable) = -1 \end{array}}{\begin{array}{c} \langle \mathcal{E}, \mathcal{S}, \{\!| m, pc, l, o, z |\!\}, \mathcal{L}, \iota, e \rangle \xrightarrow{kill} \\ \langle \mathcal{E}, \mathcal{S}, \{\!| m, pc, l, o, z |\!\}, \mathcal{L}, \iota, e \rangle \end{array}}$$

The four following rules are relative to the label transition $\epsilon$. The first rule reflects the semantics of `aload` $i$, which consists in loading an address from the local variables set of the method to its operand stack. The rule is easy and its understanding is straightforward.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{aload}\ i \\ o' = \mathsf{getLocalValue}(l, i) :: o \end{array}}{\begin{array}{c} \langle \mathcal{E}, \mathcal{S}, \{\!| m, pc, l, o, z |\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \\ \langle \mathcal{E}, \mathcal{S}, \{\!| m, pc+1, l, o', z |\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \end{array}}$$

The second rule, which uses also the label $\epsilon$, describes the semantics of the `putfield` opcode when no exceptions are signaled. The exceptions that might be thrown whenever the reference on the top of the operand stack is null, or the considered field is not found, not accessible or static. Two elements are popped from the operand stack (a value and an object reference). The considered field in the object reference is then set to the popped value.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{putfield}\ i \\ fs = \mathsf{thisConstantPoolEntry}(\mathcal{E}, m, i).fieldSignature \\ c = \mathsf{thisConstantPoolEntry}(\mathcal{E}, m, i).supposedClass \\ f = \mathsf{lookupF}(\mathcal{E}, fs, c) \\ v = \mathsf{getOneStackElem(o,0)} \\ Loc = \mathsf{getOneStackElem(o,1)} \\ (f \neq \texttt{None}) \wedge (Loc \neq \texttt{Null}) \wedge (\mathsf{accessAllowedF}(f, m)) \wedge (\neg\ \mathsf{isStaticF}(f)) \\ \mathcal{S}' = \mathcal{S}[Loc \mapsto \mathcal{S}(Loc)[fieldsMap \leftarrow \mathcal{S}(Loc).fieldsMap[f \mapsto v]]] \\ o' = \mathsf{posStack}(o, 2) \end{array}}{\begin{array}{l} \langle \mathcal{E}, \mathcal{S}, \{\!| m, pc, l, o, z |\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \longrightarrow \\ \qquad \langle \mathcal{E}, \mathcal{S}', \{\!| m, pc+1, l, o', z |\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \end{array}}$$

The third rule chosen for the label $\epsilon$ presents the semantics of `athrow`. The actual configuration moves to a configuration where the exception flag is set to the object reference on the top of the stack.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{athrow} \\ Loc = \mathsf{getOneStackElem}(o, 0) \\ (Loc \neq \texttt{Null})\ \wedge\ (\neg\mathsf{isSynchronized}(m) \vee \mathsf{isOwner}(\mathcal{S}, z, \iota)) \end{array}}{\begin{array}{l} \langle \mathcal{E}, \mathcal{S}, \{\!| m, pc, l, o, z |\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \longrightarrow \\ \qquad \langle \mathcal{E}, \mathcal{S}, \{\!| m, pc, l, o, z |\!\} :: \mathcal{F}, \mathcal{L}, \iota, Loc\rangle \end{array}}$$

The fourth rule describes the semantics of `new` $i$ in the case where the class of the object to create is not an interface, is not initialized and is not locked by another thread. A frame of its `clinit`[2] method is pushed onto the frames stack. The class is locked and added to the locked elements of the thread if it is not there yet. The monitor of the considered class is updated to reflect the fact that the current thread has acquired it or reentred it. This implies the update of the environment.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{new}\ i \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{E}, m, i) \\ \neg\ \mathsf{isInterface}(\mathcal{E}, ct) \wedge \neg\ \mathsf{isInitialized}(\mathcal{E}, ct) \\ \neg\ \mathsf{isClassLocked}(\mathcal{E}, ct) \vee \mathsf{isClassOwner}(\mathcal{E}, ct, \iota) \\ signatureClinit = \langle \texttt{clinit}, [\ ], void\rangle \\ clinit = \mathsf{retrieveM}(signatureClinit, \mathcal{E}(ct).methods) \\ clinitFrame = \mathsf{newFrame}(clinit, 0, clinit.locals, [\ ], ct) \\ \mathcal{L}' = \mathsf{ifThenElse}(\mathsf{isClassOwner}(\mathcal{E}, ct, \iota), \mathcal{L}, ct :: \mathcal{L}) \\ \mathcal{E}' = \mathcal{E}[ct \mapsto \mathsf{classMonitorEntered}(\mathcal{E}, ct, \iota)] \end{array}}{\begin{array}{l} \langle \mathcal{E}, \mathcal{S}, \{\!| m, pc, l, o, z |\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \longrightarrow \\ \qquad \langle \mathcal{E}', \mathcal{S}, clinitFrame :: \{\!| m, pc, l, o, z |\!\} :: \mathcal{F}, \mathcal{L}', \iota, \texttt{None}\rangle \end{array}}$$

[2]We consider that each class has a `clinit` method even it is empty.

The four following rules are relative to the label transition *block*. The first rule describes the semantics of `invokestatic` $i$ when the class, on which the method is invoked, is owned by another thread. The actual thread is then added to the waiting list of this class and is blocked waiting for its release.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{invokestatic}\ i \\ ms = \mathsf{thisConstantPoolEntry}(\mathcal{E}, m, i).methodSignature \\ argCount = \mathsf{length}(ms.argumentsTypes) \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{E}, m, i).supposedClass \\ \mathsf{isMethResolved}(\mathcal{E}, ms, ct) \\ m' = \mathsf{lookupM}(\mathcal{E}, ms, ct) \\ m' \neq \texttt{None} \\ cm = m'.fromClass \\ \mathsf{isSynchronized}(m') \wedge\ \mathsf{accessAllowedM}(m, m') \wedge\ \mathsf{isStaticM}(m') \wedge\ \mathsf{isInitialized}(\mathcal{E}, cm) \\ \mathsf{isClassLocked}(\mathcal{E}, cm) \wedge \neg\ \mathsf{isClassOwner}(\mathcal{E}, cm, \iota) \\ \mathcal{E}' = \mathcal{E}[cm \mapsto \mathsf{addToClassWaitingList}(\mathcal{E}, cm, \iota)] \end{array}}{\begin{array}{c} \langle \mathcal{E}, \mathcal{S}, \{\!|m, pc, l, o, z|\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \xrightarrow{block} \\ \langle \mathcal{E}', \mathcal{S}, \{\!|m, pc, l, o, z|\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \end{array}}$$

The second rule, which uses also the label *block*, reflects the semantics of the instruction `monitorenter`. It refers to the case where another thread currently owns the object referenced by the top of the operand stack. The current thread is then added to the waiting list of this object and is blocked waiting for its release.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{monitorenter} \\ Loc = \mathsf{getOneStackElem}(o, 0) \\ \mathsf{isLocked}(\mathcal{S}, Loc) \wedge \neg\ \mathsf{isOwner}(\mathcal{S}, Loc, \iota) \wedge Loc \neq \texttt{Null} \\ \mathcal{S}' = \mathcal{S}[Loc \mapsto \mathsf{addToObjectWaitingList}(\mathcal{S}, Loc, \iota)] \end{array}}{\begin{array}{c} \langle \mathcal{E}, \mathcal{S}, \{\!|m, pc, l, o, z|\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \xrightarrow{block} \\ \langle \mathcal{E}, \mathcal{S}', \{\!|m, pc, l, o, z|\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \end{array}}$$

The third rule represents the semantics of `new` $i$. It reflects the case where the relative class of the object to create is not an interface, is not initialized but is locked by another thread. The thread is then blocked waiting for the release of this class and its identity is put in the waiting list of the class.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{new}\ i \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{E}, m, i) \\ \neg\ \mathsf{isInterface}(\mathcal{E}, ct) \wedge \neg\ \mathsf{isInitialized}(\mathcal{E}, ct) \\ \mathsf{isClassLocked}(\mathcal{E}, ct) \wedge \neg\ \mathsf{isClassOwner}(\mathcal{E}, ct, \iota) \\ \mathcal{E}' = \mathcal{E}[ct \mapsto \mathsf{addToClassWaitingList}(\mathcal{E}, ct, \iota)] \end{array}}{\begin{array}{l} \langle \mathcal{E}, \mathcal{S}, \{\!|m, pc, l, o, z|\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \xrightarrow{block} \\ \qquad\qquad \langle \mathcal{E}', \mathcal{S}, \{\!|m, pc, l, o, z|\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \end{array}}$$

The fourth rule describes the semantics of `invokevirtual` $i$ when the invoked method is synchronized, accessible and not static. Furthermore, the monitor associated to the receiver is already owned by another thread. The current thread is added to the waiting list of the receiver (ie. the object referenced by the top of the current thread's operand stack) implying an update of the store. The transition label *block* transmits to the second layer the information that the current thread must be blocked.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{invokevirtual}\ i \\ ms = \mathsf{thisConstantPoolEntry}(\mathcal{E}, m, i).methodSignature \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{E}, m, i).supposedClass \\ \mathsf{isMethResolved}(\mathcal{E}, ms, ct) \\ argCount = \mathsf{length}(ms.argumentsTypes) \\ Loc = \mathsf{getOneStackElem}(o, argCount) \\ Loc \neq \texttt{Null} \\ dc = \mathsf{getDynamicClass}(\mathcal{S}, Loc) \\ m' = \mathsf{lookupM}(\mathcal{E}, ms, dc) \\ m' \neq \texttt{None} \\ \mathsf{isSynchronized}(m') \wedge\ \mathsf{accessAllowedM}(m, m') \wedge\ \neg\ \mathsf{isStaticM}(m') \\ \mathsf{isLocked}(\mathcal{S}, Loc) \wedge \neg\ \mathsf{isOwner}(\mathcal{S}, Loc, \iota) \\ \mathcal{S}' = \mathcal{S}[Loc \mapsto \mathsf{addToObjectWaitingList}(\mathcal{S}, Loc, \iota)] \end{array}}{\begin{array}{l} \langle \mathcal{E}, \mathcal{S}, \{\!|m, pc, l, o, z|\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \xrightarrow{block} \\ \qquad\qquad \langle \mathcal{E}, \mathcal{S}', \{\!|m, pc, l, o, z|\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \end{array}}$$

The four following rules use of the transition label *notify*. The first rule describes the semantics of `return` from a non-static synchronized method. We consider, in this rule, that the depth of the monitor owned by the current thread and referenced by the variable $z$ becomes null after returning from the method. We have to notify then all the threads waiting for the object referenced by $z$ and to remove the reference $z$ from the list of objects and classes locked by the current thread.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{return} \\ \mathsf{isSynchronized}(m) \wedge \neg\ \mathsf{isStaticM}(m) \wedge \mathsf{isOwner}(\mathcal{S}, z, \iota) \\ \mathcal{S}' = \mathcal{S}[z \mapsto \mathsf{objectMonitorExited}(\mathcal{S}, z, \iota)] \\ \mathsf{depthLock}(\mathcal{S}', z) = 0 \\ \mathcal{L}' = \mathsf{suppress}(z, \mathcal{L}) \\ f = \{\!|n, pc', l', o', z'|\!\} \end{array}}{\begin{array}{l} \langle \mathcal{E}, \mathcal{S}, \{\!|m, pc, l, o, z|\!\} :: f :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \xrightarrow{notify(z)} \\ \qquad \langle \mathcal{E}, \mathcal{S}', \{\!|n.pc' + 1, l', o', z'|\!\} :: \mathcal{F}, \mathcal{L}', \iota, \texttt{None}\rangle \end{array}}$$

The second rule refers also to `return` and is similar to the precedent one except that the method is static and synchronized. In this case the notification must be done for all the threads waiting for the class represented by the variable $z$ and this class is removed from the list of objects and classes locked by the current thread.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{return} \\ \mathsf{isSynchronized}(m) \wedge \mathsf{isStaticM}(m) \wedge \mathsf{isClassOwner}(\mathcal{E}, z, \iota) \\ \mathcal{E}' = \mathcal{E}[z \mapsto \mathsf{classMonitorExited}(\mathcal{E}, z, \iota)] \\ \mathsf{depthLock}(\mathcal{S}', z) = 0 \\ \mathcal{L}' = \mathsf{suppress}(z, \mathcal{L}) \\ f = \{\!|n, pc', l', o', z'|\!\} \end{array}}{\begin{array}{l} \langle \mathcal{E}, \mathcal{S}, \{\!|m, pc, l, o, z|\!\} :: f :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \xrightarrow{notify(z)} \\ \qquad \langle \mathcal{E}', \mathcal{S}, \{\!|m, pc' + 1, l', o', z'|\!\} :: \mathcal{F}, \mathcal{L}', \iota, \texttt{None}\rangle \end{array}}$$

The third rule representing the transition label *notify* describes the semantics of `monitorexit`. In this rule, the thread decrements the counter indicating the number of times the thread has entered the monitor referenced by the top of the operand stack. In this case, the counter becomes 0 and consequently, the current thread must release this monitor and notify all the threads waiting for it.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{monitorexit} \\ Loc = \mathsf{getOneStackElem}(o, 0) \\ Loc \neq \texttt{Null} \wedge \mathsf{isOwner}(\mathcal{S}, Loc, \iota) \\ \mathcal{S}' = \mathcal{S}[Loc \mapsto \mathsf{objectMonitorExited}(\mathcal{S}, Loc, \iota)] \\ \mathsf{depthLock}(\mathcal{S}', Loc) = 0 \\ \mathcal{L}' = \mathsf{suppress}(Loc, \mathcal{L}) \\ o' = \mathsf{popStack}(o, 1) \end{array}}{\begin{array}{l} \langle \mathcal{E}, \mathcal{S}, \{\!|m, pc, l, o, z|\!\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \xrightarrow{notify(Loc)} \\ \qquad \langle \mathcal{E}, \mathcal{S}', \{\!|m, pc + 1, l, o', z|\!\} :: \mathcal{F}, \mathcal{L}', \iota, \texttt{None}\rangle \end{array}}$$

The fourth rule representing the transition *notify* describes the semantics of `ireturn`. It is similar to the second rule described previously for the `return` using the label *notify* except that in the following case the method returns a value, which is pushed onto the operand stack of the previous method.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{ireturn} \\ \mathsf{isSynchronized}(m) \wedge \neg\ \mathsf{isStaticM}(m) \wedge \mathsf{isOwner}(\mathcal{S}, z, \iota) \\ \mathcal{S}' = \mathcal{S}[z \mapsto \mathsf{objectMonitorExited}(\mathcal{S}, z, \iota)] \\ \mathsf{depthLock}(\mathcal{S}', z) = 0 \\ \mathcal{L}' = \mathsf{suppress}(z, \mathcal{L}) \\ o' = \mathsf{pushStack}(o', \mathsf{head}(o)) \\ f = \{\!|n, pc', l', o', z'|\!\} \end{array}}{\begin{array}{l} \langle \mathcal{E}, \mathcal{S}, \{\!|m, pc, l, o, z|\!\} :: f :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \xrightarrow{notify(z)} \\ \qquad \langle \mathcal{E}, \mathcal{S}', \{\!|n, pc' + 1, l', o', z'|\!\} :: \mathcal{F}, \mathcal{L}', \iota, \texttt{None}\rangle \end{array}}$$

## Second Layer

The previous section described the semantics inside one thread in terms of configurations transitions: $\langle \mathcal{E}, \mathcal{S}, \mathcal{T}\rangle \xrightarrow{a} \langle \mathcal{E}, \mathcal{S}, \mathcal{T}'\rangle$ where $\mathcal{T}$ and $\mathcal{T}'$ are thread configurations and $a$ is a label in $\Lambda$. In this section, we define the semantics of multisets of threads giving the configurations transitions:$\langle \mathcal{E},\ \mathcal{S},\ \mathcal{JS}\rangle \hookrightarrow \langle \mathcal{E},\ S,\ \mathcal{JS}'\rangle$ of the unlabelled transition system $(\mathit{MultiThreadConfiguration}, \hookrightarrow)$ where $\mathcal{JS}$ and $\mathcal{JS}'$ denote Java stacks.

We present, in this section, all the rules of the second layer. There are five different rules, one for each transition label. The first rule illustrates the case when an $\varepsilon$-transition is done in the first layer. In this case, we have only to report in the Java stack the fact that the current thread has changed its information from $\mathcal{T}$ to $\mathcal{T}'$ maintaining its state to `active`. The new Java stack $\mathcal{JS}'$ is identical to the initial Java stack $\mathcal{JS}$ except for the current thread.

$$\frac{\begin{array}{c} \langle \mathcal{E},\ \mathcal{S},\ \mathcal{T}\rangle \longrightarrow \langle \mathcal{E}',\ \mathcal{S}',\ \mathcal{T}'\rangle \\ \mathcal{T}'.exception = \texttt{None} \\ \mathcal{JS}' = \mathsf{changeThreads}(\mathcal{JS}, \mathcal{T}, \mathcal{T}') \end{array}}{\langle \mathcal{E},\ S,\ \mathcal{JS} > \hookrightarrow \langle \mathcal{E}',\ \mathcal{S}',\ \mathcal{JS}'\rangle}$$

The next rule is executed when the transition in the first layer is done via the label *block*. In this case, the Java stack has to change also the state of the current thread from `active` to `blocked` because it asks for a resource that is not available.

$$\frac{\begin{array}{c} \langle \mathcal{E},\ S,\ \mathcal{T}\rangle \xrightarrow{block} \langle \mathcal{E}',\ S',\ \mathcal{T}'\rangle \\ \mathcal{JS}' = \mathsf{blockThreads}(\mathcal{JS}, \mathcal{T}) \end{array}}{\langle \mathcal{E},\ S,\ \mathcal{JS}\rangle \hookrightarrow \langle \mathcal{E}',\ S',\ \mathcal{JS}'\rangle}$$

The following rule defines the configuration transition in the second layer when the transition in the first layer is done via the label *notify*. In this case, the current thread has released an object or a class $x$ and all the threads that are waiting for this resource $x$ must be now `active`.

$$\frac{\begin{array}{c}\langle \mathcal{E},\ S,\ \mathcal{T}\rangle \xrightarrow{notify(x)} \langle \mathcal{E}',\ S',\ \mathcal{T}'\rangle \\ \mathcal{JS}' = \mathsf{changeThreads}(\mathcal{JS}, \mathcal{T}, \mathcal{T}') \\ \mathcal{JS}'' = \mathsf{activateThreads}(\mathcal{JS}', S, \mathcal{E}, [x])\end{array}}{\langle \mathcal{E},\ S,\ \mathcal{JS}\rangle \hookrightarrow \langle \mathcal{E}',\ S',\ \mathcal{JS}''\rangle}$$

The next rule presents the configuration transition in the second layer in presence of a transition in the first layer done with the label *kill*. In this case, the current thread throws an exception that is not caught by any method along its method invocation stack. The thread must then expire and activate before all the threads waiting for its locked objects.

$$\frac{\begin{array}{c}\langle \mathcal{E},\ \mathcal{S},\ \mathcal{T}\rangle \xrightarrow{kill} \langle \mathcal{E}',\ \mathcal{S}',\ \mathcal{T}'\rangle \\ \mathcal{JS}' = \mathsf{activateThreads}(\mathcal{JS}, \mathcal{S}, \mathcal{E}, \mathcal{T}.lockedElements) \\ \mathcal{JS}'' = \mathsf{dieThread}(\mathcal{JS}', \mathcal{T}.threadId)\end{array}}{\langle \mathcal{E},\ \mathcal{S},\ \mathcal{JS}\rangle \hookrightarrow \langle \mathcal{E}',\ \mathcal{S}',\ \mathcal{JS}''\rangle}$$

Finally, the last rule describes the configuration transition in the second layer when the transition in the first layer is done via the label *run*. In this case, a new thread is created and starts its execution by the adequate run method given by the lookup.

$$\frac{\begin{array}{c}\langle \mathcal{JE},\ \mathcal{S},\ \mathcal{T}\rangle \xrightarrow{run(class)} \langle \mathcal{JE}',\ \mathcal{S}',\ \mathcal{T}'\rangle \\ signatureRun = \langle run, [\ ], \mathtt{void}\rangle \\ run = \mathsf{lookupM}(\mathcal{JE}, signatureRun, class) \\ f = \mathsf{newFrame}(run, 0, run.methodVariables, [\ ], \mathtt{None}) \\ \mathcal{JS}' = \mathcal{JS}[\iota \mapsto \mathsf{newThreadInformation}([f], [\ ], \iota)\ ;\ \iota \notin Dom(\mathcal{JS})\end{array}}{\langle \mathcal{JE},\ S,\ \mathcal{JS}\rangle \hookrightarrow \langle \mathcal{JE}',\ S',\ \mathcal{JS}'\rangle}$$

## 6 CONCLUSION AND FUTURE WORK

In this paper, we reported a formalization of the dynamic semantics of JVML. The semantics comes into a small step operational style. In order to ascribe meanings to threading, the semantics is structured in two layers: The first layer capture the semantics of sequential JVML programs in isolation. The second layer consists of judgements that capture the parallel execution of JVML threads. A nice feature of the presented semantics is its faithfulness to the official JVML specification as described in [1]. Besides, the presented semantics provides full account details for the most technical and tricky aspects of JVML such as multi-threading, synchronization, method invocations, exception handling, object creation, object's fields manipulation, stack manipulation, local variable access, modifiers, etc. The presented semantics is also, to the best of our knowledge, the first dynamic semantics of JVML that provides semantics for that many features within the same framework.

As future work, we intend to use this dynamic semantics in order to explore some new and unresolved security issues in the Java virtual machine.

## REFERENCES

[1] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison Wesley, 1999.

[2] P. Bertelsen. Semantics of Java Bytecode. Technical report, Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Copenhagen, Denmark, 1997.

[3] P. Bertelsen. Dynamic Semantics of Java Bytecode. *Future Genration Computer systems*, 16(7):841–850, 2000.

[4] G. Bigliardi and C. Laneve. A Type System for JVM Threads. Technical Report UBLCS-2000-06, Department of Computer Science, University of Bologna, 2000.

[5] S. N. Freund and J. C. Mitchell. A Formal Framework for the Java Bytecode Language and Verifier. In *Proc. 14th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, volume 34(10). ACM Press, 1999.

[6] S. N. Freund and J. C. Mitchell. The Type System for Object Initialization in the Java Bytecode Language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.

[7] S. N. Freund and J. C. Mitchell. A Type System for the Java Bytecode Language and Verifier. *Journal of Automated Reasoning*, 2003.

[8] M. Hagiya and A. Tozawa. On a New Method for Dataflow Analysis of Java Virtual Machine Subroutines. In *SIG-Notes, PRO-17-3*, pages 13–18. Information Processing Society of Japan, 1998.

[9] G. Klein and M. Wildmoser. Verified bytecode subroutines. *Journal of Automated Reasoning*, 30(3–4):363–398, 2003.

[10] C. Laneve. A Type System for Jvm Threads. *Theoretical Computer Science*, 290((1)):741 – 778, 2003.

[11] I. Siveroni. Operational semantics of the java card virtual machine. *J. Log. Algebr. Program.*, 58(1-2):3–25, 2004.

[12] I. Siveroni and C. Hankin. A Proposal for the JCVMLe Operational Semantics. Technical Report SECSAFE-ICSTM-001, Imperial College,, 2001.

[13] R. Stata and M. Abadi. A Type System for Java Bytecode Subroutines. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 149–160, New York, NY, 1998.

[14] E. Borger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In *Mathematical Foundations of Computer Science*, pages 17–35, 1998.

[15] K. Havelund and K. G. Larsen. The Fork Calculus. In *ICALP*, pages 544–557, 1993.

## Appendix: Utility Functions

1. The function accessAllowedF returns true if a given field is visible from a given method:

$$\begin{array}{lcl} \mathsf{accessAllowedF} & : & \mathit{Method} \times \mathit{Field} \rightarrow \mathit{Boolean} \\ \mathsf{accessAllowedF}(m,f) & = & (\neg\ \mathsf{isPrivateF}(f) \vee (f.\mathit{fromClass}=m.\mathit{fromClass})) \end{array}$$

2. The function accessAllowedM returns true if a given method is visible from another given method:

$$\begin{array}{lcl} \mathsf{accessAllowedM} & : & \mathit{Method} \times \mathit{Method} \rightarrow \mathit{Boolean} \\ \mathsf{accessAllowedM}(m,m') & = & (\neg\ \mathsf{isPrivateM}(m') \vee (m'.\mathit{fromClass}=m.\mathit{fromClass})) \end{array}$$

3. The function activateThreads returns a new Java stack where all the threads waiting for objects or classes in a given list are activated:

$$\mathsf{activateThreads} : \mathit{JavaStack} \times \mathit{Store} \times \mathit{environment} \times (\mathit{ClassOrLocation})\texttt{-list} \rightarrow \mathit{JavaStack}$$
$$\mathsf{activateThreads}(\mathcal{JS},S,\mathcal{E},l) = \mathcal{JS}' \text{ if}$$
$$\begin{cases} \mathcal{JS}'(i) = \mathcal{JS}(i), \forall i \in \mathit{Dom}(\mathcal{JS}) \wedge i \notin \mathsf{waitingThreads}(\mathcal{S},\mathcal{E},l) \\ \mathcal{JS}'(i) = \mathsf{active}(\mathcal{JS},i), \forall i \in \mathit{Dom}(\mathcal{JS}) \wedge i \in \mathsf{waitingThreads}(S,\mathcal{E},l) \end{cases}$$

4. The function active returns the thread information of a given entry $i$ in a given *JavaStack* with an `active` state:

$$\mathsf{active} : \mathit{JavaStack} \times \mathit{Nat} \rightarrow \mathit{ThreadInformation} \times \mathit{State}$$
$$\mathsf{active}(\mathcal{JS},i) = (\mathcal{JS}(i).\mathit{threadInformation}, \texttt{active})$$

5. The function addToClassWaitingList adds a given thread in the waiting list of a given class:

$$\mathsf{addToClassWaitingList}: \mathit{Environment} \times \mathit{ClassType} \times \mathit{ThreadId} \rightarrow \mathit{Class}$$
$$\mathsf{addToClassWaitingList}(\mathcal{E},ct,id) = \mathcal{E}(ct)[\mathit{monitorClass.waitList} \leftarrow \mathrm{id}::\mathcal{E}(ct).\mathit{monitorClass.waitList}]$$

6. The function addToObjectWaitingList adds a given thread in the waiting list of a given object:

$$\mathsf{addToObjectWaitingList}: \mathit{Store} \times \mathit{Location} \times \mathit{ThreadId} \rightarrow \mathit{JavaObject}$$
$$\mathsf{addToObjectWaitingList}(S,Loc,id) = S(Loc)[\mathit{monitor.waitList} \leftarrow \mathrm{id}::S(loc).\mathit{monitor.waitList}]$$

7. The function allInterfaces gives all the interfaces of a given set of classes.

$$
\begin{array}{lcl}
\mathsf{allInterfaces} & : & \mathit{Environment} \times (\mathit{ClassType})\text{-}\mathtt{set} \rightarrow (\mathit{ClassType})\text{-}\mathtt{set} \\
\mathsf{allInterfaces}(\mathcal{E},\emptyset) & = & \emptyset \\
\mathsf{allInterfaces}(\mathcal{E},\{ct\}) & = & \emptyset \text{ if } ct = \text{“}\mathtt{Object}\text{”} \\
\mathsf{allInterfaces}(\mathcal{E},\{ct\}) & = & \mathcal{E}(ct).\mathit{interfaces} \\
 & & \cup\ \mathsf{allInterfaces}(\mathcal{E},\{\mathcal{E}(ct).\mathit{superClass}\}) \\
 & & \cup\ \mathsf{allSuperClasses}(\mathcal{E},\mathcal{E}(ct).\mathit{interfaces}) \text{ if } ct \neq \text{“}\mathtt{Object}\text{”} \\
\mathsf{allInterfaces}(\mathcal{E},\{ct\} \cup \mathit{ctSet}) & = & \mathsf{allInterfaces}(\mathcal{E},\{ct\}) \\
 & & \cup\ \mathsf{allInterfaces}(\mathcal{E},\mathit{ctSet}) \text{ if } \mathit{ctSet} \neq \emptyset
\end{array}
$$

8. The function allSuperClasses gives all the super classes of a given set of classes.

$$
\begin{array}{l}
\mathsf{allSuperClasses} : \mathit{Environment} \times (\mathit{ClassType})\text{-}\mathtt{set} \rightarrow (\mathit{ClassType})\text{-}\mathtt{set} \\
\mathsf{allSuperClasses}(\mathcal{E},\emptyset) = \emptyset \\
\mathsf{allSuperClasses}(\mathcal{E},\{ct\}) = \emptyset \text{ if } ct = \text{“}\mathtt{Object}\text{”} \\
\mathsf{allSuperClasses}(\mathcal{E},\{ct\}) = \\
\qquad \mathcal{E}(ct).\mathit{superClass} \cup \mathsf{allSuperClasses}(\mathcal{E},\{\mathcal{E}(ct).\mathit{superClass}\}) \text{ if } ct \neq \text{“}\mathtt{Object}\text{”} \\
\mathsf{allSuperClasses}(\mathcal{E},\{ct\} \cup \mathit{ctSet}) = \mathsf{allSuperClasses}(\mathcal{E},\{ct\}) \cup \mathsf{allSuperClasses}(\mathcal{E},\mathit{ctSet})
\end{array}
$$

9. The function appropriatePcHandler returns the start address indicated by the first appropriate exception handler found given an environment, a program counter, an exception class type to catch and a list of exception handlers. If there is no appropriate exception handler in the list, the function return $-1$:

$$
\begin{array}{ll}
\mathsf{appropriatePcHandler} & : \mathit{Environment} \times \mathit{Nat} \times \mathit{ClassType} \times \mathit{ExceptionTable} \rightarrow \mathtt{int} \\
\mathsf{appropriatePcHandler}(\mathcal{E}, pc, ct, [\,]) & = -1 \\
\mathsf{appropriatePcHandler}(\mathcal{E}, pc, ct, l) & = \mathsf{head}(l).\mathit{handler} \\
 & \quad \text{if } \mathsf{isAppropriateHandler}(\mathcal{E}, pc, ct, \mathsf{head}(l)) \\
\mathsf{appropriatePcHandler}(\mathcal{E}, pc, ct, l) & = \mathsf{appropriatePcHandler}(\mathcal{E}, pc, ct, \mathsf{tail}(l)) \\
 & \quad \text{if } \neg\ \mathsf{isAppropriateHandler}(\mathcal{E}, pc, ct, \mathsf{head}(l))
\end{array}
$$

10. The function blockThreads sets the state of a thread to `blocked` in the Java stack:

$$
\begin{array}{l}
\mathsf{blockThreads} : \mathit{JavaStack} \times \mathit{ThreadInformation} \rightarrow \mathit{JavaStack} \\
\mathsf{blockThreads}(\mathcal{JS},\mathcal{T}) = \mathcal{JS}' \text{ if } \left\{ \begin{array}{l}
\mathcal{T}.\mathit{threadId} = \mathit{id} \\
\mathcal{JS}'(i) = \mathcal{JS}(i), \forall i \in \mathit{Dom}(\mathcal{JS}) \wedge i \neq \mathit{id} \\
\mathcal{JS}'(\mathit{id}).\mathit{state} = \mathtt{blocked} \\
\mathcal{JS}'(\mathit{id}).\mathit{threadInformation} = \mathcal{T}
\end{array} \right.
\end{array}
$$

11. The function changeThreads allows to change a given thread information to another given thread information in a given Java stack and maintaining the state to `active`:

$$\mathsf{changeThreads}:\ \mathit{JavaStack} \times \mathit{ThreadInformation} \times \mathit{ThreadInformation} \rightarrow \mathit{JavaStack}$$

$$\mathsf{changeThreads}(\mathcal{JS},\mathcal{T},\mathcal{T}') = \mathcal{JS}' \text{ if } \begin{cases} \mathcal{T}.\mathit{threadId} = \mathit{id} \\ \mathcal{JS}'(i) = \mathcal{JS}(i), \forall i \in \mathit{Dom}(\mathcal{JS}) \wedge i \neq \mathit{id} \\ \mathcal{JS}'(\mathit{id}).\mathit{state} = \mathcal{JS}(\mathit{id}).\mathit{state} \\ \mathcal{JS}'(\mathit{id}).\mathit{threadInformation} = \mathcal{T}' \end{cases}$$

12. The function **classMonitorEntered** returns the same class than the given class identifier but containing the information that the class's monitor has been entered by a given thread:

$$\begin{array}{lcl} \mathsf{classMonitorEntered} & : & \mathit{Environment} \times \mathit{ClassType} \times \mathit{ThreadId} \rightarrow \mathit{Class} \\ \mathsf{classMonitorEntered}(\mathcal{E},\mathit{ct},\mathit{id}) & = & \end{array}$$

$$\begin{array}{lll} \mathcal{E}(\mathit{ct})[ & \mathit{monitorClass.waitList} & \leftarrow \mathsf{suppress}(\mathcal{E}(\mathit{ct}).\mathit{monitorClass.waitList},\mathrm{id}), \\ & \mathit{monitorClass.threadOwner} & \leftarrow \mathit{id}, \\ & \mathit{monitorClass.depth} & \leftarrow \mathcal{E}(\mathit{ct}).\mathit{monitorClass.depth}{+}1] \end{array}$$

13. The function **classMonitorExited** returns the same class than the given class identifier but containing the information that the class's monitor has been exited by a given thread:

$$\begin{array}{lcl} \mathsf{classMonitorExited} & : & \mathit{Environment} \times \mathit{ClassType} \times \mathit{ThreadId} \rightarrow \mathit{Class} \\ \mathsf{classMonitorExited}(\mathcal{E},\mathit{ct},\mathit{id}) & = & \end{array}$$

$$\begin{array}{lll} \mathcal{E}(\mathit{ct}) & [\mathit{monitorClass.threadOwner} & \leftarrow \text{“}\texttt{None}\text{”} /\ \mathcal{E}(\mathit{ct}).\mathit{monitorClass.depth}{=}1, \\ & \mathit{monitorClass.depth} & \leftarrow \mathcal{E}(\mathit{ct}).\mathit{monitorClass.depth}{-}1] \end{array}$$

14. The function **depthLock** returns the number of times an object in a given Location and a given store has been reentered. If the object is not owned by any thread, the value returned is zero:

$$\begin{array}{lcl} \mathsf{depthLock} & : & \mathit{Store} \times \mathit{Location} \rightarrow \mathit{Nat} \\ \mathsf{depthLock}(S,\mathit{loc}) & = & (S(\mathit{loc}).\mathit{monitor}).\mathit{depth} \end{array}$$

15. The function **dieThread** returns a Java stack constructed by removing a given thread from a given Java stack:

$$\mathsf{dieThread}:\ \mathit{JavaStack} \times \mathit{ThreadId} \rightarrow \mathit{JavaStack}$$

$$\mathsf{dieThread}(\mathcal{JS},\mathit{id}) = \mathcal{JS}' \text{ if } \begin{cases} \mathit{Dom}(\mathcal{JS}') = \mathit{Dom}(\mathcal{JS}) - \{\mathit{id}\} \\ \forall i \in \mathit{Dom}(\mathcal{JS}'); \mathcal{JS}'(i) = \mathcal{JS}(i) \end{cases}$$

16. The function **getDynamicClass** returns the dynamic class of an object at a given location in a given store:

$$\begin{array}{lll} \mathsf{getDynamicClass} & : & Store \times Location \rightarrow ClassType \\ \mathsf{getDynamicClass}(S,loc) & = & S(loc).classType \end{array}$$

17. The function **getLocalValue** returns an element at a given position in a given list:

$$\begin{array}{lll} \mathsf{getLocalValue} & : & (\tau)\text{-}\mathsf{list} \times Nat \rightarrow \tau \\ \mathsf{getLocalValue}(l,0) & = & \mathsf{head}(l) \\ \mathsf{getLocalvalue}(l,i) & = & \mathsf{getLocalValue}(\mathsf{tail}(l),i\text{-}1),\ \forall\ i > 0 \end{array}$$

18. The function **getOneStackElem** is identical to **getLocalValue**. It has been introduced only to let the comprehension of the semantic rules easier:

$$\begin{array}{lll} \mathsf{getOneStackElem} & : & (\tau)\text{-}\mathsf{list} \times Nat \rightarrow \tau \\ \mathsf{getOneStackElem}(l,i) & = & \mathsf{getLocalValue}(l,i) \end{array}$$

19. The function **head** returns the first element in a given list:

$$\begin{array}{lll} \mathsf{head} & : & (\tau)\text{-}\mathsf{list} \rightarrow \tau \\ \mathsf{head}(v::l) & = & v,\ \forall\ (v,l) \in \tau \times (\tau)\text{-}\mathsf{list} \end{array}$$

20. The function **ifThenElse** returns, depending on the value of a given boolean value, the second or the third given argument:

$$\begin{array}{ll} \mathsf{ifThenElse} & : Boolean \times a \times a \rightarrow a \\ \mathsf{ifThenElse}(\mathtt{true},a,b) & = a \\ \mathsf{ifThenElse}(\mathtt{false},a,b) & = b \end{array}$$

21. The function **isAppropriateHandler** returns true if a given exception handler is appropriate for a given program counter and a thrown exception in a specific environment otherwise the function returns false:

$$\begin{array}{ll} \mathsf{isAppropriateHandler} & : Environment \times Nat \times ClassType \times ExceptionHandler \rightarrow Boolean \\ \mathsf{isAppropriateHandler}(\mathcal{E},pc,ct,h) & = \mathtt{True}\ \text{if} \left\{ \begin{array}{l} h.\mathsf{startPc} <= pc \\ h.\mathsf{endPc} >= pc \\ h.\mathsf{exceptionType} \in \mathsf{allSuperClasses}(\mathcal{E},ct) \end{array} \right. \\ \mathsf{isAppropriateHandler}(\mathcal{E},pc,ct,h) & = \mathtt{false}\ \text{otherwise.} \end{array}$$

22. The function isClassOwner returns true if in a given environment, a given thread Id is owner of a given class otherwise the function returns false:

$$\begin{array}{lcl} \mathsf{isClassOwner} & : & \mathit{Environment}\times \mathit{ClassType}\times \mathit{ThreadId} \rightarrow \mathit{Boolean} \\ \mathsf{isClassOwner}(\mathcal{E},c,id) & = & ((\mathcal{E}(c).\mathit{monitorClass}).\mathit{threadOwner}= \mathrm{id}) \end{array}$$

23. The function isInitialized returns true if in a given environment, a given class is initialized otherwise the function returns false:

$$\begin{array}{lcl} \mathsf{isInitialized} & : & \mathit{Environment} \times \mathit{ClassType} \rightarrow \mathit{Boolean} \\ \mathsf{isInitialized}(\mathcal{E},c) & = & (\mathcal{E}(c).\mathit{initialized}=1) \end{array}$$

24. The function isInterface returns true if in a given environment, a given class is an interface otherwise the function returns false

$$\begin{array}{lcl} \mathsf{isInterface} & : & \mathit{Environment} \times \mathit{ClassType} \rightarrow \mathit{Boolean} \\ \mathsf{isInterface}(\mathcal{E},c) & = & (\mathcal{E}(c).\mathit{interface}=1) \end{array}$$

25. The function isLocked returns true if in a given store, a given location is locked otherwise the function returns false:

$$\begin{array}{lcl} \mathsf{isLocked} & : & \mathit{Store}\times \mathit{Location} \rightarrow \mathit{Boolean} \\ \mathsf{isLocked}(S,loc) & = & (S(loc).\mathit{monitor}).\mathit{threadOwner} \neq \text{“}\mathtt{None}\text{”}) \end{array}$$

26. The function isMethodOf returns true if a class with an identifier belonging to a given list of class identifiers contains a method with the same signature than the given method signature otherwise the function returns false:

$$\mathsf{isMethodOf} : \mathit{Environment}\times \mathit{MethodSignature}\times (\mathit{ClassType})\text{-}\mathtt{set} \rightarrow \mathit{Boolean}$$

$$\begin{array}{lcl} \mathsf{isMethodOf}(\mathcal{E},ms,\emptyset) & = & \mathsf{false} \\ \mathsf{isMethodOf}(\mathcal{E},ms,\{ct\}) & = & \mathsf{true\ if}\ ms \in \mathsf{methodSignatures}(\mathcal{E}(ct).\mathit{methods}) \\ \mathsf{isMethodOf}(\mathcal{E},ms,\{ct\}) & = & \mathsf{false\ if}\ ms \notin \mathsf{methodSignatures}(\mathcal{E}(ct).\mathit{methods}) \\ \mathsf{isMethodOf}(\mathcal{E},ms,\{ct\} \cup ctSet) & = & \mathsf{isMethodOf}(\mathcal{E},ms,\{ct\}) \vee \mathsf{isMethodOf}(\mathcal{E},ms,ctSet) \end{array}$$

27. The function isMethResolved returns true if a given method signature is resolved for a given class in a given environment otherwise the function returns `false`:[3]

[3] A method $m$ is resolved for a class $c$ [1] if $c$ is not an interface and $c$, or one of the super classes of $c$ or any of the superinterfaces of $c$ declares $m$.

$$
\begin{array}{lcl}
\mathsf{isMethResolved} & : & \mathit{Environment} \times \mathit{MethodSignature} \times \mathit{ClassType} \rightarrow \mathit{Boolean} \\
\mathsf{isMethResolved}(\mathcal{E},ms,ct) & = & \text{false if } \mathsf{isInterface}(\mathcal{E},ct) \\
\mathsf{isMethResolved}(\mathcal{E},ms,ct) & = & \text{false if} \begin{cases} \neg\mathsf{isInterface}(\mathcal{E},ct) \\ \mathit{classes1} = \{ct\} \cup \mathsf{allSuperClasses}(\mathcal{E},\{ct\}) \\ \mathit{classes}{=}\mathit{classes1} \cup \mathsf{allInterfaces}(\mathcal{E},\{ct\}) \\ \neg\mathsf{isMethodOf}(\mathcal{E},ms,classes) \end{cases} \\
\mathsf{isMethResolved}(\mathcal{E},ms,ct) & = & \text{true if} \begin{cases} \neg\mathsf{isInterface}(\mathcal{E},ct) \\ \mathit{classes1} = \{ct\} \cup \mathsf{allSuperClasses}(\mathcal{E},\{ct\}) \\ \mathit{classes}{=}\mathit{classes1} \cup \mathsf{allInterfaces}(\mathcal{E},\{ct\}) \\ \mathsf{isMethodOf}(\mathcal{E},ms,classes) \end{cases}
\end{array}
$$

28. The function isOwner returns true if an object in a given store and pointed by a given Location is acquired by a thread with a given Id otherwise the function returns false:

$$
\begin{array}{lcl}
\mathsf{isOwner} & : & \mathit{Store} \times \mathit{Location} \times \mathit{ThreadId} \rightarrow \mathit{Boolean} \\
\mathsf{isOwner}(S,loc,id) & = & ((S(loc).monitor).threadOwner{=}\ id)
\end{array}
$$

29. The function isPrivateF returns true if a given field is private otherwise the function returns false:

$$
\begin{array}{lcl}
\mathsf{isPrivateF} & : & \mathit{Field} \rightarrow \mathit{Boolean} \\
\mathsf{isPrivateF}(f) & = & (\mathsf{private} \in f.\mathit{fieldModifiers})
\end{array}
$$

30. The function isPrivateM returns true if a given method is private otherwise the function returns false:

$$
\begin{array}{lcl}
\mathsf{isPrivateM} & : & \mathit{Method} \rightarrow \mathit{Boolean} \\
\mathsf{isPrivateM}(m) & = & (\ \mathsf{private} \in m.\mathit{methodModifiers})
\end{array}
$$

31. The function isStaticF returns true if a given field is static otherwise the function returns false:

$$
\begin{array}{lcl}
\mathsf{isStaticF} & : & \mathit{Field} \rightarrow \mathit{Boolean} \\
\mathsf{isStaticF}(f) & = & (\mathsf{static} \in f.\mathit{fieldModifiers})
\end{array}
$$

32. The function isStaticM returns true if a given method is static otherwise the function returns false:

$$
\begin{array}{lcl}
\mathsf{isStaticM} & : & \mathit{Method} \rightarrow \mathit{Boolean} \\
\mathsf{isStaticM}(m) & = & (\mathsf{static} \in m.\mathit{methodmMdifiers})
\end{array}
$$

33. The function isSynchronized returns true if a given method is synchronized otherwise the function returns false:

$$\begin{array}{lcl} \mathsf{isSynchronized} & : & \mathit{Method} \rightarrow \mathit{Boolean} \\ \mathsf{isSynchronized}(m) & = & (\mathsf{synchronized} \in m.\mathit{methodModifiers}) \end{array}$$

34. The function isThread returns true if a given class type is the class `Thread` or one of its subclasses otherwise the function returns false:

$$\begin{array}{lcl} \mathsf{isThread} & : & \mathit{Environment} \times \mathit{ClassType} \rightarrow \mathit{Boolean} \\ \mathsf{isThread}(\mathcal{E},ct) & = & (ct{=}\texttt{Thread}) \vee (\texttt{Thread} \in \mathsf{allSuperClasses}(\mathcal{E},ct)) \end{array}$$

35. The function length returns the length of a given list:

$$\begin{array}{lcl} \mathsf{length} & : & (\tau)\text{-}\mathsf{list} \rightarrow \mathit{Nat} \\ \mathsf{length}([\,]) & = & 0 \\ \mathsf{length}(v{::}l) & = & 1 + \mathsf{length}\ (l),\ \forall\ (v,l)\ \in \tau\times(\tau)\text{-}\mathsf{list} \end{array}$$

36. The function lookupF returns the first field with a given signature found in the superclass hierarchy of a given class in a given environment. If the field is not found the value `None` is returned:

$$\mathsf{lookupF}\text{: } \mathit{Environment}\times \mathit{FieldSignature}\times \mathit{ClassType} \rightarrow \mathit{Field} \bigoplus \mathit{NoneType}$$
$$\mathsf{lookupF}(\mathcal{E},fs,c) = f \text{ if } \mathsf{retrieveF}(fs,\mathcal{E}(\mathrm{c}).\mathit{fields}){=}f \wedge f \neq \text{“}\texttt{None}\text{”}$$
$$\mathsf{lookupF}(\mathcal{E},fs,c) = \mathsf{lookupF}(\mathcal{E},fs,\mathcal{E}(c).\mathit{superClass}) \text{ if } \begin{cases} \mathrm{c}{\neq} \text{“}\texttt{Object}\text{”} \\ \mathsf{retrieveF}(\mathrm{fs}, \mathcal{E}(\mathrm{c}).\mathit{fields}) = \text{“}\texttt{None}\text{”} \end{cases}$$
$$\mathsf{lookupF}(\mathcal{E},fs,c) = \text{“}\texttt{None}\text{”} \quad \text{if} \begin{cases} \mathrm{c}{=} \text{“}\texttt{Object}\text{”} \\ \mathsf{retrieveF}(\mathrm{fs}, \mathcal{E}(\mathrm{c}).\mathit{fields}) = \text{“}\texttt{None}\text{”} \end{cases}$$

37. The function lookupM returns the first method with a given signature found in the superclass hierarchy of a given class in a given environment. If the method is not found the value `None` is returned:

$$\mathsf{lookupM} : \mathit{Environment}\times \mathit{MethodSignature}\times \mathit{ClassType} \rightarrow \mathit{Method} \bigoplus \mathit{NoneType}$$
$$\mathsf{lookupM}(\mathcal{E},ms,c) = m \text{ if } \mathsf{retrieveM}(ms,\mathcal{E}(\mathrm{c}).\mathit{methods}){=}m \wedge m \neq \text{“}\texttt{None}\text{”}$$
$$\mathsf{lookupM}(\mathcal{E},ms,c) = \mathsf{lookupM}(ms,\mathcal{E}(c).\mathit{superClass}) \text{ if } \begin{cases} \mathrm{c}{\neq} \text{“}\texttt{Object}\text{”} \\ \mathsf{retrieveM}(\mathrm{ms}, \mathcal{E}(\mathrm{c}).\mathit{methods}) = \text{“}\texttt{None}\text{”} \end{cases}$$
$$\mathsf{lookupM}(\mathcal{E},ms,c) = \text{“}\texttt{None}\text{”} \quad \text{if} \begin{cases} \mathrm{c}{=} \text{“}\texttt{Object}\text{”} \\ \mathsf{retrieveM}(\mathrm{ms}, \mathcal{E}(\mathrm{c}).\mathit{methods}) = \text{“}\texttt{None}\text{”} \end{cases}$$

38. The function newFrame returns a new frame constructed from a given method, a given program counter, a given list of local variable values a given operand stack and a given synchronized element:

$$\mathsf{newFrame}\colon Method\times ProgramCounter\times Locals\times OperandStack\times SynchronizedElement\rightarrow Frame$$

$$\mathsf{newFrame}(m,pc,l,o,z) \;=\; frame \;\text{ if } \begin{cases} frame.method = \mathrm{m} \\ frame.programCounter = \mathrm{pc} \\ frame.locals = \mathrm{l} \\ frame.operandStack = \mathrm{o} \\ frame.synchronizedElement = \mathrm{z} \end{cases}$$

39. The function newThreadInformation returns a new thread structure given a thread stack, a list of locked elements and a thread Id:

$$\mathsf{newThreadInformation}\colon ThreadStack\times (LockedElement)\text{-}\mathit{list}\times ThreadId \rightarrow ThreadInformation$$

$$\mathsf{newThreadInformation}(s,l,id) = t \;\text{ if } \begin{cases} t.threadStack = \mathrm{s} \\ t.lockedElements = \mathrm{l} \\ t.threadId = \mathrm{id} \end{cases}$$

40. The function objectMonitorExited returns the same object than the given location in the given store but containing the information that the object's monitor has been exited by a given thread:

$$\begin{array}{lcl} \mathsf{objectMonitorExited} & : & Store\times Location\times ThreadId \rightarrow JavaObject \\ \mathsf{objectMonitorExited}(S,Loc,id) & = & S(Loc)[monitor.threadOwner \leftarrow \text{“}\mathtt{None}\text{”}/ \\ & & \qquad S(Loc).monitor.depth{=}1, \\ & & monitor.depth \qquad \leftarrow S(Loc).monitor.depth\text{-}1] \end{array}$$

41. The function posStack returns a list obtained from a given list by popping a given number of elements:

$$\begin{array}{lcl} \mathsf{posStack} & : & (\tau)\text{-list} \times Nat \rightarrow (\tau)\text{-list} \\ \mathsf{posStack}(l,0) & = & l \\ \mathsf{popStack}(l,i) & = & \mathsf{popStack}(\mathsf{tail}(l),i\text{-}1)),\ \forall\ i > 0 \end{array}$$

42. The function pushStack returns a list obtained by appending a given value to a given list:

$$\begin{array}{lcl} \mathsf{pushStack} & : & (\tau)\text{-list}\times\tau \rightarrow (\tau)\text{-list} \\ \mathsf{pushStack}(l,v) & = & \mathrm{v::l},\ \forall\ (v,l) \in \tau\times(\tau)\text{-list} \end{array}$$

43. The function retrieveF searches for a field with a given signature in a given list of fields and returns the field if the field is found otherwise returns the value `None`:

$$\begin{array}{lcl} \mathsf{retrieveF} & : & \mathit{FieldSignature} \times \mathit{Fields} \rightarrow \mathit{Field} \oplus \mathit{NoneType} \\ \mathsf{retrieveF}(fs,[\,]) & = & \text{"}\texttt{None}\text{"} \\ \mathsf{retrieveF}(fs,l) & = & \mathsf{head}(l) \text{ if } \mathsf{head}(l).\mathit{fieldSignature} = \mathrm{fs} \\ \mathsf{retrieveF}(fs,l) & = & \mathsf{retrieveF}(fs,\mathsf{tail}(l)) \text{ if } \mathsf{head}(l).\mathit{fieldSignature} \neq \mathrm{fs} \end{array}$$

44. The function retrieveM searches for a method with a given signature in a given list of methods and returns the method if the method has been found otherwise returns the value `None`:

$$\begin{array}{lcl} \mathsf{retrieveM} & : & \mathit{MethodSignature} \times \mathit{Methods} \rightarrow \mathit{Method} \oplus \mathit{NoneType} \\ \mathsf{retrieveM}(ms,[\,]) & = & \text{“}\texttt{None}\text{”} \\ \mathsf{retrieveM}(ms,l) & = & \mathsf{head}(l) \text{ if } \mathsf{head}(l).\mathit{methodSignature} = \mathrm{ms} \\ \mathsf{retrieveM}(ms,l) & = & \mathsf{retrieveM}(ms,\mathsf{tail}(l)) \text{ if } \mathsf{head}(l).\mathit{methodSignature} \neq \mathrm{ms} \end{array}$$

45. The function suppress returns a list constructed from a given list by suppressing all the occurrences of a given value :

$$\begin{array}{lcl} \mathsf{suppress} & : & \tau \times (\tau)\text{-list} \rightarrow (\tau)\text{-set} \\ \mathsf{suppress}(\mathrm{v},[\,]) & = & [\,] \\ \mathsf{suppress}(\mathrm{v},\mathrm{l}) & = & \mathsf{suppress}(\mathrm{v},\mathsf{tail}(\mathrm{l})); \text{ if } \mathsf{head}(\mathrm{l})=\mathrm{v} \\ \mathsf{suppress}(\mathrm{v},\mathrm{l}) & = & \mathsf{head}(\mathrm{l})::\mathsf{suppress}(\mathrm{v},\mathsf{tail}(\mathrm{l})); \text{ if } \mathsf{head}(\mathrm{l}) \neq \mathrm{v} \end{array}$$

46. The function tail returns the tail of a given list:

$$\begin{array}{lcl} \mathsf{tail} & : & (\tau)\text{-list} \rightarrow (\tau)\text{-list} \\ \mathsf{tail}([\,]) & = & [\,] \\ \mathsf{tail}(v::l) & = & l, \forall\ (v,l) \in \tau \times (\tau)\text{-list} \end{array}$$

47. The function thisConstantPoolEntry returns a constant pool entry given an environment, a method and an index for the entry:

$$\begin{array}{lcl} \mathsf{thisConstantPoolEntry} & : & \mathit{Environment} \times \mathit{Method} \times \mathit{Nat} \rightarrow \mathit{ConstantPoolEntry} \\ \mathsf{thisConstantPoolEntry}(\mathcal{E},m,i) & = & \mathcal{E}(m.\mathit{fromClass}).\mathit{constantPool}(i) \end{array}$$

48. This function waitingThreads returns a set of thread Ids waiting for a list of objects or classes described in a given list of locations or class identifiers:

$$
\begin{array}{l}
\mathsf{waitingThreads} : \mathit{Store}\times\mathit{Environment}\times\texttt{(}\mathit{ClassOrLocation}\texttt{)-list} \rightarrow \texttt{(}\mathit{ThreadId}\texttt{)-set}\\
\mathsf{waitingThreads}(\mathcal{S},\mathcal{E},[\,]) = \emptyset\\
\mathsf{waitingThreads}(\mathcal{S},\mathcal{E},\mathrm{x::l}) = \mathsf{setOf}(\mathrm{S(x)}.\mathit{waitList}) \cup\\
\qquad\qquad \mathsf{waitingThreads}(\mathcal{S},\mathcal{E},\mathsf{tail}(\mathrm{l})) \text{ if } \mathrm{x} \in \mathrm{Dom(S)}\\
\mathsf{waitingThreads}(\mathcal{S},\mathcal{E},\mathrm{x::l}) = \mathsf{setOf}(\mathcal{E}(\mathrm{x}).\mathit{monitor.waitList}) \cup\\
\qquad\qquad \mathsf{waitingThreads}(\mathcal{S},\mathcal{E},\ \mathsf{tail}(\mathrm{l})) \text{ if } \mathrm{x} \in \mathcal{E}(\mathrm{S})
\end{array}
$$

## ABOUT THE AUTHORS

**Nadia Belblidia** is a Ph.D. student at Concordia Institute for Information Systems Engineering (CIISE), Concordia University, Montreal, Quebec, Canada. She is a member of the Computer Security Laboratory (CSL) at CIISE. She is pursuing a Ph.D. thesis on an aspect oriented approach for security hardening. She can be reached at na_bel@ece.concordia.ca.

**Mourad Debbabi** Mourad Debbabi is a Full Professor and the Associate Director of the Concordia Institute for Information Systems Engineering at Concordia University. He is also a Concordia Research Chair Tier I in Information Systems Security. He holds Ph.D. and M.Sc. degrees in computer science from Paris-XI Orsay, University, France. He published more than 100 research papers in international journals and conferences on computer security, formal semantics, mobile and embedded platforms, Java technology security and acceleration, cryptographic protocol specification, design and analysis, malicious code detection, programming languages, type theory and specification and verification of safety-critical systems. In the past, he served as Senior Scientist at the Panasonic Information and Network Technologies Laboratory, Princeton, New Jersey, USA; Associate Professor at the Computer Science Department of Laval University, Quebec, Canada; Senior Scientist at General Electric Corporate Research Center, New York, USA; Research Associate at the Computer Science Department of Stanford University, California, USA; and Permanent Researcher at the Bull Corporate Research Center, Paris, France. Nowadays, he is leading two major projects on cyber forensics and open source security.