Syntax of the JCVM Language
To Be Studied in the SecSafe Project

Date: 2001/05/16 15:37:19 (UTC)
Authors: Renaud Marlet (Trusted Logic)
Classification: Public
Number: SECSAFE-TL-005
Version: 1.7
Status: Stable

Abstract. This report describes the syntax of the JCVML*e* language, that models the Java Card Virtual Machine Language. We propose JCVML*e* as the language to be studied in the SecSafe project.

JCVML*e* is presented in a style that allows an easy browsing of linked programs. The goal of this formalization is to provide a high-level representation that abstracts uninteresting language details while preserving the generality and applicability to realistic Java Card programs.

Contents

1 Introduction **2**
- 1.1 The Framework . . . 2
- 1.2 The Language Subset Issue . . . 3
- 1.3 Case Studies and Implementation . . . 3
- 1.4 Organization of This Document . . . 3
- 1.5 Notations . . . 3

2 Program Structure **5**
- 2.1 Program . . . 5
- 2.2 Package . . . 5
- 2.3 Class . . . 6
- 2.4 Interface . . . 6
- 2.5 Method . . . 7
- 2.6 Field . . . 8
- 2.7 Type . . . 8
- 2.8 Concrete Syntax . . . 9

3 Instructions **10**
- 3.1 Operand Type . . . 10
- 3.2 Instruction Set . . . 10
- 3.3 Concrete Syntax . . . 13

References **18**

A Brief Comparison With Freund and Mitchell's Formalization **19**

B Linking Issues **21**

1 Introduction

Java Card programs are compiled into the Java Card Virtual Machine Language (JCVML) [Sun00, Sun99]. This reports describes JCVML*e*, a language that captures the *e*ssence of JCVML, abstracting minor language details, while preserving the full expression power of the JCVM language. We propose JCVML*e* as the language to be studied in the SecSafe project. We call JCVM*e* the JCVM variant supporting the language JCVML*e*.

The structure of JCVML*e* is presented as functions operating over a data structure (e.g., to traverse the class hierarchy): a program in JCVML*e* can be thought of as a graph and most functions merely follow an edge of the graph from one node to another one, e.g., from a class to its superclass to traverse the class hierarchy. Although JCVML*e* is expressed in terms of functions, it merely is a syntactic representation; this representation is rich and abstract though. A more traditional concrete syntax is also provided. This concrete syntax is intended only as a practical communication means between humans; it is not intended to be parsed by any tool (see section 1.3).

1.1 The Framework

The framework described in this document hides uninteresting language and VM details so that the focus can be kept on the salient constructs and features of the JCVM. Program analyses and semantics are to be expressed on this core JCVML*e* language, rather than on actual JCVM programs. Tools will provides the link between both visions (see §1.3).

CAP File Format Abstraction. The format of the JCVM CAP files are substantially different from the JVM class files, especially regarding linking information. In the JCVM, linking is based on *tokens* and *offsets* rather than names and types, as in the JVM — in fact, CAP file translation gets rid of all names[1]. Tokens are small numbers identifying externally visible (i.e., public or protected) program items. Tokens are not global identifiers; they are meaningful only in the scope of a given package or class. Tokens are also used as indexes into various tables in the CAP file as well as runtime data structures. Some indirect lookups into tables may require doing some (simple) arithmetic on tokens. Linkable program items that are not externally visible (i.e., private or package-visible) are referred to using direct offsets into CAP file components rather than tokens. The mapping from Java names and types to tokens and offsets is performed during CAP file conversion [Sun00].

A major abstraction of our framework relates to the format of the CAP file. All offsets into CAP file components, all tokens, and all table lookups (including cascades of lookups to search for an item) are hidden. The program structure made available provides direct access to all information and allows easy program browsing; all instructions have direct access to the relevant data structure. Only purely dynamic linking, required for virtual and interface method invocation, has to be retained. It is made as abstract as possible though, and close to the JVM class member identification scheme. See appendix B for more details.

High-level Instructions. Moreover, similar JCVML instructions have been grouped in JCVML*e*. The resulting high-level instructions possibly take extra parameters. (See also appendix A for a comment on instruction grouping compared to Freund and Mitchell's formalization.) Here are some typical examples:

- all numeric operations such as `add` and `sub` are represented using a single `numop` instruction,

[1] The only strings representing names that remain are externally visible names located in the Export files.

- `load_0` is not retained as it is just a special case of `load`,
- `new` now encompasses the `newarray` and `anewarray`,
- `invoke` factorizes `invokestatic` and `invokespecial`, etc.

All in all, the 185 JCVML instructions have thus been reduced to 30 in JCVML*e*. It must be noted that JCVML*e* retains the full expression power of the JCVM. As a matter of fact, there is a "direct" translation from JCVML*e* programs to JCVML programs, and conversely. This translation does not require any analysis; it is somehow "syntactic".

1.2 The Language Subset Issue

One important thing to keep in mind is that the closest we stay to the original JCVM, the more guarantee we have that our analyses will provide results that we can rely on. As a matter of fact, this issue was somehow mentioned in [MLM00] regarding the code of bytecode vs. source code[2].

As said above, JCVML*e* captures all JCVML. This does not imply that all JCVML*e* instructions and features have to be supported right from the beginning of the SecSafe project. Rather, they should be thought of as final targets and participants could possibly focus on different subsets in the course of the project. In other words, which language subset is treated at a given time of the project is a separate issue, not addressed in this document. A proposition of a hierarchy of language features to be studied in SecSafe can be found in [Mar00].

1.3 Case Studies and Implementation

This formalization will be implemented as a library providing functions to read and link CAP files, and to traverse the program structure. This will include all the functions listed in sections 2 and 3. An analyzer, or any other semantic tool, will just have to interface with this library.

Java Card case studies that Trusted Logic will provide will use the CAP file format. Reading such files using the library will provide access to the high-level representation. There will not be any intermediate, syntactic representation of JCVML*e*, to be later parsed by an analyzer or any other semantic tool.

1.4 Organization of This Document

The report is organized as follows. Section 2 describes the general structure of a program. Section 3 presents the different instructions. Appendix A briefly compares our framework to Freund and Mitchell's formalization [FM99].

1.5 Notations

The formalization presented in this report is expressed in an algebraic form. We use the following notations. (Capitalization conventions roughly follows those of Java programming.)

[2] "Bytecode should be considered rather than Java or Java Card source language. This is especially important in the context of certifications at the highest levels of the Common Criteria because proving properties on source code would not be considered as sufficient. An extra proof would be required to ensure that the desired properties are preserved by the translation into bytecode. This translation is not straightforward in the case of Java Card (Java compilation and CAP file generation); a two-phase verification process would be significantly more complex than a direct bytecode verification."

Domains. What we call a *domain* is this document merely is a set equipped with functions. Domains are written using slanted capitalized names, e.g., *Class*. If the name is made of several words, each of them is capitalized, e.g., *ReturnAddressType*.

Primitive domains, which also correspond to primitive values of the Java language, are not capitalized though, e.g., *boolean*, *int*. We also borrow the *String* domain from Java.

A $\perp$ subscript denotes a domain augmented with an extra $\perp$ element, typically to represent an error or an unknown value, e.g., $Class_\perp$.

Constants. Constants are written with slanted uppercase names, e.g., *TRUE*. If the name is made of several words, they are uppercase and separated by an underscore, e.g., *RETURN_ADDRESS*.

Constant of primitive domains are not uppercase though, e.g., *true*, *false*. Strings are written between double quotes, e.g., *"java.lang.Object"*.

Functions. Functions are written using slanted lowercase names, e.g., $type : Field \to Type$. If the name is made of several words, they are all capitalized except the first one which is lowercase, e.g., $isPackageVisible : Field \to boolean$. Primitive domains and constants also have slanted lowercase names though (see above).

Function Overloading. Different functions sometimes have the same name; in this case, their type is used to disambiguate them. This overloading — common in mathematics — simplifies the correspondence between the formalization and the implementation, which will be written in Java and use method overloading.

Variables. Variables and placeholders are written with italicized names, e.g., *index*, X. Variables are sometimes used in function declarations to disambiguate domains. E.g., function

$$lookupTable : KeySwitchInstruction \to [\, int \times int \,]$$

is actually declared as:

$$lookupTable : KeySwitchInstruction \to [\, (match : int) \times (offset : int) \,]$$

Variables can also be used to clarify the use of a domain, for instance to specify a measure unit, e.g.,

$$maxOperandStackHeight : BytecodeMethod \to (nbWords : int)$$

Program Elements. Java and JCVM program elements are written using a typewriter font, e.g., Java keyword `try`, Java and JCVM type `byte`, JCVM instruction `sadd`.

Domain Inclusion, Disjointness and Membership. Except otherwise noted, all domains defined in this document are disjoints. In a domain definition, disjoint union is stressed using the notation $A = B_1 \mid \ldots \mid B_n$, meaning $A = B_1 \cup \ldots \cup B_n$ and $\forall i \neq j, B_i \cap B_j = \varnothing$. E.g., $Type = PrimitiveType \mid ReferenceType$. This can be somehow thought of as a sum domain.

It is possible to test the membership of a domain, e.g., "if $t \in ReferenceType$ then ...", as if there were a function

$$isA : Element \times Domain \to boolean$$

Access Functions. Functions listed in section 2 provide a functional access to data structures representing the program. To simplify notations and get rid of a few parentheses — and keeping up with the spirit of object orientation — we sometimes write $x.f$ instead of $f(x)$. For instance, the methods of a class c can be referred to as $c.\mathit{methods}$ instead of $\mathit{methods}(c)$. Likewise, testing if a method m is static can be written $m.\mathit{isStatic}$ instead of $\mathit{isStatic}(m)$. This can actually apply to any function with one argument.

When a function has more than one argument, the remaining arguments can be mentioned as in object-oriented languages, i.e., $x_1.f(x_2, \ldots, x_n)$ instead of $f(x_1, x_2, \ldots, x_n)$. This notation does not reduce the size of the expression; it is only used to put a stress on the first argument, especially when it represents a kind of environment for computing f. For example, the class named n in a program p can be noted $p.\mathit{class}(n)$ instead of $\mathit{class}(p, n)$.

Arrays. For any set X, we note $[\,X\,]$ the set of finite arrays of elements of X. The length of an array can be known using function $\mathit{length} : [\,X\,] \to \mathit{int}$. The indices of an array a range from 0 to $a.\mathit{length} - 1$.

Sets. For any set X, we note $\{\!|\,X\,|\!\}$ the set of finite sets of elements of X. The size (cardinality) of a finite set can be known using function $\mathit{size} : \{\!|\,X\,|\!\} \to \mathit{int}$.

Ranges. A range (interval) of integers between values a and b (inclusive) is written $a\,..\,b$.

2 Program Structure

We use the term *program structure* to refer to all program information besides instructions. It includes in particular the class hierarchy and is indispensable for expressing the semantics of a JCVML*e* program.

2.1 Program

A *program* contains a set of packages. A class, interface or package can be retrieved from a program given a fully-qualified name. It is also possible to retrieve a package given its AID (see §2.2).

$$\begin{array}{rcl}
\mathit{class} & : & \mathit{Program} \times (\mathit{longName} : \mathit{String}) \to \mathit{Class}_\perp \\
\mathit{interface} & : & \mathit{Program} \times (\mathit{longName} : \mathit{String}) \to \mathit{Interface}_\perp \\
\mathit{package} & : & \mathit{Program} \times (\mathit{longName} : \mathit{String}) \to \mathit{Package}_\perp \\
\mathit{package} & : & \mathit{Program} \times \mathit{AID} \to \mathit{Package}_\perp \\
\mathit{packages} & : & \mathit{Program} \to \{\!|\,\mathit{Package}\,|\!\}
\end{array}$$

The $\perp$ subscript indicates that looking up for a class, interface or package, given a name, may fail.

2.2 Package

A *package* contains classes and interfaces. It has an AID as well as major and minor version numbers.

$$\begin{array}{rcl}
\mathit{classes} & : & \mathit{Package} \to \{\!|\,\mathit{Class}\,|\!\} \\
\mathit{interfaces} & : & \mathit{Package} \to \{\!|\,\mathit{Interface}\,|\!\}
\end{array}$$

$$\begin{aligned} \mathit{aid} &: \mathit{Package} \to \mathit{AID} \\ \mathit{majorVersion} &: \mathit{Package} \to \mathit{int} \\ \mathit{minorVersion} &: \mathit{Package} \to \mathit{int} \end{aligned}$$

An AID (application identifier) consists of an array of bytes. It can be tested for equality.

$$\begin{aligned} \mathit{bytes} &: \mathit{AID} \to [\,\mathit{byte}\,] \\ \mathit{equals} &: \mathit{AID} \times \mathit{AID} \to \mathit{boolean} \end{aligned}$$

2.3 Class

A *class* belongs to a package. It has a (direct) superclass as well as (direct) subclasses. It (directly) implements interfaces. It has (direct) fields and methods; inherited (indirect) fields and methods have to be looked up in the superclass. It has so-called "access flags".

$$\begin{aligned} \mathit{package} &: \mathit{Class} \to \mathit{Package} \\ \mathit{superClass} &: \mathit{Class} \to \mathit{Class}_{\perp} \\ \mathit{subClasses} &: \mathit{Class} \to \{\!|\,\mathit{Class}\,|\!\} \\ \mathit{implementedInterfaces} &: \mathit{Class} \to \{\!|\,\mathit{Interface}\,|\!\} \\ \mathit{fields} &: \mathit{Class} \to \{\!|\,\mathit{Field}\,|\!\} \\ \mathit{methods} &: \mathit{Class} \to \{\!|\,\mathit{Method}\,|\!\} \\ \mathit{isFinal} &: \mathit{Class} \to \mathit{boolean} \\ \mathit{isPublic} &: \mathit{Class} \to \mathit{boolean} \\ \mathit{isAbstract} &: \mathit{Class} \to \mathit{boolean} \end{aligned}$$

The possible $\perp$ value for *superClass* corresponds to the fact that all classes have a superclass, except `java.lang.Object` that has none.

2.4 Interface

An *interface* belongs to a package. It (directly) extends superinterfaces and it is (directly) extended by subinterfaces. It is (directly) implemented by classes. It contains (direct) fields and methods. It has "access flags".

$$\begin{aligned} \mathit{package} &: \mathit{Interface} \to \mathit{Package} \\ \mathit{subInterfaces} &: \mathit{Interface} \to \{\!|\,\mathit{Interface}\,|\!\} \\ \mathit{superInterfaces} &: \mathit{Interface} \to \{\!|\,\mathit{Interface}\,|\!\} \\ \mathit{implementingClasses} &: \mathit{Interface} \to \{\!|\,\mathit{Class}\,|\!\} \\ \mathit{fields} &: \mathit{Interface} \to \{\!|\,\mathit{Field}\,|\!\} \\ \mathit{methods} &: \mathit{Interface} \to \{\!|\,\mathit{Method}\,|\!\} \\ \mathit{isFinal} &: \mathit{Interface} \to \mathit{boolean} \\ \mathit{isPublic} &: \mathit{Interface} \to \mathit{boolean} \\ \mathit{isAbstract} &: \mathit{Interface} \to \mathit{boolean} \\ \mathit{isShareable} &: \mathit{Interface} \to \mathit{boolean} \end{aligned}$$

2.5 Method

A *method* has a type and its arguments represents a known number of words on the operand stack. It has "access flags". It knows in which class or interface it is defined or declared.

$$\begin{aligned}
\mathit{type} &: \mathit{Method} \to \mathit{MethodType} \\
\mathit{nbArgumentWords} &: \mathit{Method} \to \mathit{int} \\
\mathit{isInit} &: \mathit{Method} \to \mathit{boolean} \\
\mathit{isAbstract} &: \mathit{Method} \to \mathit{boolean} \\
\mathit{isPackageVisible} &: \mathit{Method} \to \mathit{boolean} \\
\mathit{isProtected} &: \mathit{Method} \to \mathit{boolean} \\
\mathit{isPrivate} &: \mathit{Method} \to \mathit{boolean} \\
\mathit{isPublic} &: \mathit{Method} \to \mathit{boolean} \\
\mathit{isFinal} &: \mathit{Method} \to \mathit{boolean} \\
\mathit{isStatic} &: \mathit{Method} \to \mathit{boolean} \\
\mathit{classI} &: \mathit{Method} \to \mathit{Class} \cup \mathit{Interface}
\end{aligned}$$

Moreover, each method has a *method ID*, used for virtual and interface method invocation. The equality test on method IDs is the smallest equivalence relation that satisfies the following properties.

- If a method overrides another one, their method IDs are equal.
- If an instance method implements an interface method, their method IDs are equal.

In other cases (besides equivalence based on these two properties), two method IDs are considered different.

$$\begin{aligned}
\mathit{id} &: \mathit{Method} \to \mathit{MethodID} \\
\mathit{equals} &: \mathit{MethodID} \times \mathit{MethodID} \to \mathit{boolean}
\end{aligned}$$

A method that is not abstract is a *bytecode method*, that has additional information concerning the method body. In particular, the bytecode is decoded and linked, forming a sequence of *Instruction*s (see §3). These instructions are accessed via (abstract) addresses. The following data is available : address of the first instruction; address of the instruction following the instruction at a given address; instruction at a given address. And it is possible to check if an address precedes another one in a method (this is used for exception handling). A bytecode method also has an array of exception handlers. It knows the maximum number of words needed to store local variables when executing the method, as well as the maximum operand stack height (number of words).

$$\mathit{BytecodeMethod} \subset \mathit{Method}$$

$$\begin{aligned}
\mathit{firstAddress} &: \mathit{BytecodeMethod} \to \mathit{Address} \\
\mathit{nextAddress} &: \mathit{BytecodeMethod} \times \mathit{Address} \to \mathit{Address}_{\perp} \\
\mathit{instructionAt} &: \mathit{BytecodeMethod} \times \mathit{Address} \to \mathit{Instruction} \\
\mathit{precedes} &: \mathit{BytecodeMethod} \times \mathit{Address} \times \mathit{Address} \to \mathit{boolean} \\
\mathit{maxOperandStackHeight} &: \mathit{BytecodeMethod} \to (\mathit{nbWords} : \mathit{int}) \\
\mathit{maxLocalVariableArraySize} &: \mathit{BytecodeMethod} \to (\mathit{nbWords} : \mathit{int}) \\
\mathit{exceptionHandlers} &: \mathit{BytecodeMethod} \to [\,\mathit{ExceptionHandler}\,]
\end{aligned}$$

An *exception handler* is structured as follows. It has a catch type, that is undefined for "`finally`" handlers. It has bytecode addresses to mark the beginning and end (inclusively) of the "`try`" region. It has an address corresponding to the code that handles the exception. The exception handlers ordering is the same as specified in [LY99, Sun00].

$$\begin{aligned} \mathit{catchType} &: \mathit{ExceptionHandler} \to \mathit{ClassType}_{\perp} \\ \mathit{startAddress} &: \mathit{ExceptionHandler} \to \mathit{Address} \\ \mathit{endAddress} &: \mathit{ExceptionHandler} \to \mathit{Address} \\ \mathit{handlerAddress} &: \mathit{ExceptionHandler} \to \mathit{Address} \end{aligned}$$

2.6 Field

A *field* has a type. It has "access flags". It knows in which class or interface it is defined or declared. A static field possibly has an initial value. (Initial values are made explicit in the CAP file format of the JCVM, as opposed to class files of the JVM, that initialize static fields using static initializers, i.e., `<clinit>` methods.)

$$\begin{aligned} \mathit{type} &: \mathit{Field} \to \mathit{Type} \\ \mathit{isPackageVisible} &: \mathit{Field} \to \mathit{boolean} \\ \mathit{isProtected} &: \mathit{Field} \to \mathit{boolean} \\ \mathit{isPrivate} &: \mathit{Field} \to \mathit{boolean} \\ \mathit{isPublic} &: \mathit{Field} \to \mathit{boolean} \\ \mathit{isFinal} &: \mathit{Field} \to \mathit{boolean} \\ \mathit{isStatic} &: \mathit{Field} \to \mathit{boolean} \\ \mathit{classI} &: \mathit{Field} \to \mathit{Class} \cup \mathit{Interface} \\ \mathit{initValue} &: \mathit{Field} \to \mathit{int} \cup [\,\mathit{int}\,] \cup \{\mathit{null}, \perp\} \end{aligned}$$

Moreover, each field has a unique *field ID* that can be tested for equality.

$$\begin{aligned} \mathit{id} &: \mathit{Field} \to \mathit{FieldID} \\ \mathit{equals} &: \mathit{FieldID} \times \mathit{FieldID} \to \mathit{boolean} \end{aligned}$$

2.7 Type

A *method type* represents the argument types as well as the result type of a method.

$$\begin{aligned} \mathit{parameterTypes} &: \mathit{MethodType} \to [\,\mathit{Type}\,] \\ \mathit{resultType} &: \mathit{MethodType} \to \mathit{ResultType} \end{aligned}$$

There are different kinds of *types*.

$$\begin{aligned} \mathit{Type} &= \mathit{ReferenceType} \mid \mathit{PrimitiveType} \\ \mathit{ReferenceType} &= \mathit{ArrayType} \mid \mathit{ClassType} \mid \mathit{InterfaceType} \\ \mathit{PrimitiveType} &= \mathit{BooleanType} \mid \mathit{NumericType} \mid \mathit{ReturnAddressType} \\ \mathit{NumericType} &= \mathit{IntegralType} \\ \mathit{IntegralType} &= \mathit{ByteType} \mid \mathit{ShortType} \mid \mathit{IntType} \\ \mathit{ResultType} &= \mathit{Type} \mid \mathit{VoidType} \end{aligned}$$

There is a single representative for each different primitive type as well as for the void type. An undetermined reference type is also provided; it is used to annotate instructions with the general kind of value it operates on.

$$\begin{aligned} \mathit{ReturnAddressType} &= \{\mathit{RETURN\_ADDRESS}\} \\ \mathit{BooleanType} &= \{\mathit{BOOLEAN}\} \\ \mathit{ByteType} &= \{\mathit{BYTE}\} \\ \mathit{ShortType} &= \{\mathit{SHORT}\} \\ \mathit{IntType} &= \{\mathit{INT}\} \\ \mathit{VoidType} &= \{\mathit{VOID}\} \\ \mathit{ReferenceType} &\supset \{\mathit{REFERENCE}\} \end{aligned}$$

A type can be tested for subtyping. It has a number of dimensions, which is null (zero) if it is not an array type — this is not to be confused with the length of a one-dimension array.

$$\begin{aligned} \mathit{isSubtypeOf} &: \mathit{Type} \times \mathit{Type} \to \mathit{boolean} \\ \mathit{nbDimensions} &: \mathit{Type} \to \mathit{int} \end{aligned}$$

In addition, class and interface types provide access to the corresponding class and interface. An array type provides access to its element type.

$$\begin{aligned} \mathit{class} &: \mathit{ClassType} \to \mathit{Class} \\ \mathit{interface} &: \mathit{InterfaceType} \to \mathit{Interface} \\ \mathit{elementType} &: \mathit{ArrayType} \to \mathit{Type} \end{aligned}$$

2.8 Concrete Syntax

The above formalization of JCVML*e* programs uses a functional style. It actually gives access to a rich underlying data structure that is graph, rather than a tree. It is thus not easy to visualize or to communicate, especially via a screen or a piece of paper.

We define here a concrete syntax for JCVML*e* programs. This grammar is intended only for comprehension and communication between humans. In the context of the project, there is no interest in building a parser for it.

As JCVML*e* programs actually come from Java (Card) programs, we decided to reuse a large part of the Java grammar. One major advantage of this solution is that it is easy to understand to anybody that knows Java. Moreover, it naturally captures all the features that are relevant for the JCVM. The only difference relates to method and fields IDs, which we do not detail here: a field ID is to be thought of as a qualified field name and a method ID as a method signature (in the Java sense, i.e., name and argument types, see [LY99, GJSB00]).

In practice, we build upon the grammar for Java defined in [GJSB00]. All program items corresponding to packages, classes, interfaces, methods, fields and types are borrowed verbatim from this grammar. We only need to redefine the grammar rule that defines a Java method body (see [GJSB00, section 8.4.5]); we define it here in terms of JCVML*e* instructions rather than Java statements. (Alternatives in a grammar rule are written as separate lines. See [GJSB00, chapter 2] for a complete specification of the syntax of grammar rules.)

MethodBody:
InstructionBlock
;

Instruction blocks are defined in section 3.3, which also provides a example of concrete syntax for a complete program.

3 Instructions

Instructions are decoded from arrays of bytecode found in bytecode method bodies (see §2.5). This involves resolving references that are possibly present among their static parameters, such as references to class, fields, etc.

3.1 Operand Type

As Java VM instructions, most JCVML*e* instruction are typed: they operate on values that have an expected type. Possible *instruction operand types* are the following.

$$\mathit{OperandType} \quad = \quad \mathit{NumericType} \cup \{\mathit{REFERENCE}\}$$

NumericType and *REFERENCE* are defined in §2.7. The *REFERENCE* type represents any reference type. It also indicates a possible return address in the case of the *StoreLocalVariableInstruction* (used for subroutine calls). The *BYTE* type (element of *NumericType*) is used by the Java compiler to also implement booleans; there is no *BOOLEAN* in *OperandType*. Only field and method types can explicitly refer to type `boolean`.

Most instructions only make sense for a subset of these operand types. However, for readability reasons, the instruction signature given in the following section does not try to enforce operand typing. Conversely, some types are used only by a limited number of instructions. The extreme case in the *VOID* type, that is only used by the *ReturnInstruction*.

Note that the operand type is not needed for the operational semantics of JCVML*e*. This information, present in the original bytecode, is only used to make "defensive" VM that do not have a bytecode verifier. As we assume the bytecode to be verified, the operand type is just informative.

3.2 Instruction Set

JCVML*e* instructions are the following.

$$\begin{array}{rcl} \mathit{Instruction} & = & \mathit{ArrayLengthInstruction} \\ & | & \mathit{ArrayLoadInstruction} \\ & | & \mathit{ArrayStoreInstruction} \\ & | & \mathit{CheckCastInstruction} \\ & | & \mathit{ConditionalBranchInstruction} \\ & | & \mathit{DuplicateWordsInstruction} \\ & | & \mathit{GetInstanceFieldInstruction} \\ & | & \mathit{GetStaticFieldInstruction} \\ & | & \mathit{IncrementLocalVariableInstruction} \end{array}$$

| *IndexSwitchInstruction*
| *InstanceOfInstruction*
| *InvokeDefiniteMethodInstruction*
| *InvokeInterfaceMethodInstruction*
| *InvokeVirtualMethodInstruction*
| *JumpSubroutineInstruction*
| *KeySwitchInstruction*
| *LoadLocalVariableInstruction*
| *NewInstruction*
| *NoOperationInstruction*
| *NumericOperationInstruction*
| *PopWordsInstruction*
| *PushConstantValueInstruction*
| *PutInstanceFieldInstruction*
| *PutStaticFieldInstruction*
| *ReturnInstruction*
| *ReturnSubroutineInstruction*
| *StoreLocalVariableInstruction*
| *SwapWordsInstruction*
| *ThrowInstruction*
| *UnconditionalBranchInstruction*

These instructions have the following parameters.

type : *ArrayLoadInstruction* → *OperandType*
type : *ArrayStoreInstruction* → *OperandType*
type : *CheckCastInstruction* → *ReferenceType*
operation : *ConditionalBranchInstruction* → *NumericComparison*
operandType : *ConditionalBranchInstruction* → *OperandType*
nullComparison : *ConditionalBranchInstruction* → *boolean*
target : *ConditionalBranchInstruction* → *Address*
nbWords : *DuplicateWordsInstruction* → *byte*
insertionDepth : *DuplicateWordsInstruction* → *byte*
field : *GetInstanceFieldInstruction* → *Field*
fromThis : *GetInstanceFieldInstruction* → *boolean*
field : *GetStaticFieldInstruction* → *Field*
type : *IncrementLocalVariableInstruction* → *OperandType*
index : *IncrementLocalVariableInstruction* → *int*
constant : *IncrementLocalVariableInstruction* → *short*
type : *IndexSwitchInstruction* → *OperandType*

$$
\begin{array}{rcll}
\mathit{lowValue} & : & \mathit{IndexSwitchInstruction} & \rightarrow \mathit{int} \\
\mathit{targets} & : & \mathit{IndexSwitchInstruction} & \rightarrow [\,\mathit{Address}\,] \\
\mathit{defaultTarget} & : & \mathit{IndexSwitchInstruction} & \rightarrow \mathit{Address} \\
\mathit{type} & : & \mathit{InstanceOfInstruction} & \rightarrow \mathit{ReferenceType} \\
\mathit{method} & : & \mathit{InvokeDefiniteMethodInstruction} & \rightarrow \mathit{Method} \\
\mathit{baseMethod} & : & \mathit{InvokeInterfaceMethodInstruction} & \rightarrow \mathit{Method} \\
\mathit{baseMethod} & : & \mathit{InvokeVirtualMethodInstruction} & \rightarrow \mathit{Method} \\
\mathit{target} & : & \mathit{JumpSubroutineInstruction} & \rightarrow \mathit{Address} \\
\mathit{type} & : & \mathit{KeySwitchInstruction} & \rightarrow \mathit{OperandType} \\
\mathit{lookupTable} & : & \mathit{KeySwitchInstruction} & \rightarrow [\,(\mathit{match} : \mathit{int}) \times (\mathit{address} : \mathit{Address})\,] \\
\mathit{defaultTarget} & : & \mathit{KeySwitchInstruction} & \rightarrow \mathit{Address} \\
\mathit{type} & : & \mathit{LoadLocalVariableInstruction} & \rightarrow \mathit{OperandType} \\
\mathit{index} & : & \mathit{LoadLocalVariableInstruction} & \rightarrow \mathit{int} \\
\mathit{type} & : & \mathit{NewInstruction} & \rightarrow \mathit{ReferenceType} \\
\mathit{operation} & : & \mathit{NumericOperationInstruction} & \rightarrow \mathit{NumericOperation} \\
\mathit{operandType} & : & \mathit{NumericOperationInstruction} & \rightarrow \mathit{OperandType} \\
\mathit{resultType} & : & \mathit{NumericOperationInstruction} & \rightarrow \mathit{OperandType} \\
\mathit{nbWords} & : & \mathit{PopWordsInstruction} & \rightarrow \mathit{byte} \\
\mathit{constant} & : & \mathit{PushConstantInstruction} & \rightarrow \{\mathit{null}\} \cup \mathit{int} \\
\mathit{type} & : & \mathit{PushConstantInstruction} & \rightarrow \mathit{OperandType} \\
\mathit{field} & : & \mathit{PutInstanceFieldInstruction} & \rightarrow \mathit{Field} \\
\mathit{toThis} & : & \mathit{PutInstanceFieldInstruction} & \rightarrow \mathit{boolean} \\
\mathit{field} & : & \mathit{PutStaticFieldInstruction} & \rightarrow \mathit{Field} \\
\mathit{resultType} & : & \mathit{ReturnInstruction} & \rightarrow \mathit{OperandType} \cup \mathit{VoidType} \\
\mathit{index} & : & \mathit{ReturnSubroutineInstruction} & \rightarrow \mathit{int} \\
\mathit{type} & : & \mathit{StoreLocalVariableInstruction} & \rightarrow \mathit{OperandType} \\
\mathit{index} & : & \mathit{StoreLocalVariableInstruction} & \rightarrow \mathit{int} \\
\mathit{nbTopWords} & : & \mathit{SwapWordsInstruction} & \rightarrow \mathit{byte} \\
\mathit{nbWordsBelow} & : & \mathit{SwapWordsInstruction} & \rightarrow \mathit{byte} \\
\mathit{target} & : & \mathit{UnconditionalBranchInstruction} & \rightarrow \mathit{Address}
\end{array}
$$

As indicated in the JCVM specification, a *StoreLocalVariableInstruction* that is marked as operating on a value of reference type can actually also operate on a return address. Numeric operations and comparisons are as follows.

$$
\begin{array}{rcl}
\mathit{NumericComparison} & = & \{\mathit{EQ}, \mathit{GE}, \mathit{GT}, \mathit{LE}, \mathit{LT}, \mathit{NE}\} \\
\mathit{NumericOperation} & = & \mathit{UnaryNumericOperation} \cup \mathit{BinaryNumericOperation} \\
\mathit{UnaryNumericOperation} & = & \{\mathit{NEG}, \mathit{TO}\} \\
\mathit{BinaryNumericOperation} & = & \{\mathit{ADD}, \mathit{SUB}, \mathit{MUL}, \mathit{DIV}, \mathit{REM}, \mathit{CMP}, \\
 & & \mathit{AND}, \mathit{OR}, \mathit{XOR}, \mathit{SHL}, \mathit{SHR}, \mathit{USHR}\}
\end{array}
$$

The numeric operation *TO* expresses numeric type conversion. E.g., the `i2s` instruction corresponds to a *NumericOperationInstruction* i such that $i.operandType = INT$, $i.resultType = SHORT$, $i.operation = TO$.

3.3 Concrete Syntax

As explained in section 2.8, we express the concrete syntax of a JCVML*e* program on the basis of the grammar for Java defined in [GJSB00]. As is the case for most grammars, not all parsable programs have a meaning; semantic restrictions are those of the instruction parameters, whose functional declaration was given in the preceding section.

Instruction Block. An instruction block represents an implemented method body. It consists of a list of instructions labeled with an address, and a list of exception handlers. Although the functional representation relies on relative offsets, the concrete syntax is expressed in terms of absolute addresses (i.e., offset from 0) to make reading easier; the correspondence is straightforward. An exception handler is given by an "active" range of addresses, a class type (or `*` if this is a "`finally`" handler) to match and an address to jump to in case of matching.

InstructionBlock:
`{` *LabeledInstructions*$_{opt}$ *ExceptionHandlers*$_{opt}$ `}`

LabeledInstructions:
LabeledInstruction
LabeledInstructions LabeledInstruction

LabeledInstruction:
Address `:` *Instruction*

ExceptionHandlers:
ExceptionHandler
ExceptionHandlers ExceptionHandler

ExceptionHandler:
Address `-` *Address* `:` *CatchType* `=>` *Address*

CatchType:
ClassType
`*`

Nonterminals *ClassType* and *DecimalNumeral* are defined in the grammar for Java [GJSB00].

Addresses. Addresses appear as numbers in the concrte syntax. However, they should be thought of as labels because there is no arithmetic on addresses: all offsets are turned into "absolute" addresses.

Address:
DecimalNumeral

The instructions of a method body are labeled with successive address numbers, starting at 0. In the operational semantics rules, instead of writing "$m.$*nextAddress*(pc)" or "*nextAddress*(m, pc)", it can be convenient to only write "$pc + 1$". (The method m is understood.) It is just a matter of explicitly introducing the appropriate notation.

Instructions. Instruction mnemonics are based on those found in the JVM and JCVM. The type prefix used in most JVM and JCVM instruction has been dropped and turned into an explicit parameter, following the functional representation of instructions. We use the letter "`r`" instead of "`a`" to indicate a reference or, possibly, a return address in the case of the `store` instruction (used for subroutine calls).

Instruction:

`nop`
`push` *OperandType Constant*
`pop` *NbWords*
`dup` *NbWords NbWords*
`swap` *NbWords NbWords*
`numop` *OperandType NumericOperator* $OperandType_{opt}$
`load` *OperandType LocalVariableIndex*
`store` *OperandType LocalVariableIndex*
`inc` *OperandType LocalVariableIndex IntegerConstant*
`goto` *Address*
`if` *ComparisonOperator OperandType* $NullComparison_{opt}$ `goto` *Address*
`lookupswitch` *OperandType* $MatchTableEntries_{opt}$ `, default =>` *Address*
`tableswitch` *OperandType IndexTableEntry* `, default =>` *Address*
`new` *ReferenceType*
`checkcast` *ReferenceType*
`instanceof` *ReferenceType*
`getstatic` *Name*
`putstatic` *Name*
`getfield` $\texttt{this}_{opt}$ *Name*
`putfield` $\texttt{this}_{opt}$ *Name*
`invokedefinite` *Signature*
`invokevirtual` *Signature*
`invokeinterface` *Signature*
`return` $OperandType_{opt}$
`arraylength`
`arrayload` *OperandType*
`arraystore` *OperandType*
`throw`
`jsr` *Address*
`ret` *LocalVariableIndex*

OperandType: one of

`r b s i`

Constant:

NullLiteral
IntegerConstant

IntegerConstant:

$\texttt{-}_{opt}$ *IntegerLiteral*

NbWords:

DecimalNumeral

ComparisonOperator: one of
`eq ne gt le lt ne`

NullComparison: one of
`0 null`

NumericOperator: one of
`neg add sub mul div rem and or xor shl shr ushr to cmp`

LocalVariableIndex:
DecimalNumeral

MatchTableEntries:
MatchTableEntry
MatchTableEntries `,` *MatchTableEntry*

MatchTableEntry:
IntegerConstant `=>` *Address*

IndexTableEntry:
IntegerConstant `=>` $Addresses_{opt}$

Addresses:
Address
Addresses Address

Signature:
Name `(` $TypeList_{opt}$ `)`

TypeList:
Type
Type `,` *TypeList*

The *OperandType* nonterminal represent respectively the reference (or, possibly, return address) type and the types `byte` (or `boolean`), `short` and `int`.

The *NullComparison* is optional. If present, it indicates a conditional that compares the operand stack top with *0* or *null*. E.g., "`if eq s 0`" models "`ifeq`". If absent, the comparison is between the two top elements of the operand stack. E.g., "`if eq s`" models "`if_scmpeq`".

The keyword `this` is optional in instructions `getfield` and `putfield`. It allows the representation of JCVM specialized instructions. E.g., "`getfield this`" models "`getfield_this_r`" (if the field is of type reference).

Nonterminals *ReferenceType*, *NullLiteral*, *IntegerLiteral*, *DecimalNumeral* and *Name* are defined in the grammar for Java [GJSB00]. The nonterminals *ReferenceType* and *ResultType* are not to be confused with the domains *ReferenceType* and *ResultType*, defined in section 2.7.

Table 1 informally relates the concrete syntax to the functional representation. It is also convenient as a summary of the JCVML*e* instruction set. Figure 1 provides a concrete syntax example.

Instruction	Mnemonic	Effect
NoOperationInstruction	`nop`	No operation
PushConstantInstruction	`push` t c	Push constant c of type t
PopWordsInstruction	`pop` n	Pop n top words
DuplicateWordsInstruction	`dup` m n	Duplicate words $1 .. m$ at depth n
SwapWordsInstruction	`swap` m n	Swap words $1 .. m$ with words $m{+}1 .. m{+}n$
NumericOperationInstruction	`numop` t *op* t'_{opt}	Numeric operation *op*, operand typed t, result t' (if $\neq t$)
LoadLocalVariableInstruction	`load` t i	Load from local variable of type t at index i
StoreLocalVariableInstruction	`store` t i	Store into local variable of type t at index i
IncrementLocalVariableInstruction	`inc` t i c	Increment local variable of type t at index i by constant c
UnconditionalBranchInstruction	`goto` a	Branch at address a
ConditionalBranchInstruction	`if` t *op* *nullCmp* `goto` a	Branch at a if typed t operand(s) compare w.r.t. *op*
KeySwitchInstruction	`lookupswitch` t $(k$ `=>` $a)^*$, `default =>` a'	Branch at a_i if typed t operand $= k_i$, otherwise at a'
IndexSwitchInstruction	`tableswitch` t l `=>` $(a)^*$, `default =>` a'	Branch at a_i if typed t operand $= l + i$, otherwise at a'
NewInstruction	`new` t	Create new object (possibly array) of reference type t
CheckCastInstruction	`checkcast` t	Check whether object is of reference type t
InstanceOfInstruction	`instanceof` t	Determine if object is of reference type t
GetStaticFieldInstruction	`getstatic` f	Get static field f from class
PutStaticFieldInstruction	`putstatic` f	Put static field f in class
GetInstanceFieldInstruction	`getfield` $\texttt{this}_{opt}$ f	Fetch field f from (possibly this) object
PutInstanceFieldInstruction	`putfield` $\texttt{this}_{opt}$ f	Set field f in (possibly this) object
InvokeDefiniteMethodInstruction	`invokedefinite` m	Invoke definite (static or special) method m
InvokeVirtualMethodInstruction	`invokevirtual` m	Invoke virtual method m
InvokeInterfaceMethodInstruction	`invokeinterface` m	Invoke interface method m
ReturnInstruction	`return` t_{opt}	Return `void` or value typed t from method
ArrayLengthInstruction	`arraylength`	Get length of array
ArrayLoadInstruction	`arrayload` t	Load value from array of type t
ArrayStoreInstruction	`arraystore` t	Store value into array of type t
ThrowInstruction	`throw`	Throw exception or error
JumpSubroutineInstruction	`jsr` a	Jump to subroutine at address a
ReturnSubroutineInstruction	`ret` i	Return from subroutine to address in local variable i

Table 1: Instruction set summary

```
package p;

class Num { short val; }

public abstract class C
{
  protected short rescue;
  // No explicit contructor for C




  public short abs(Num n)
  {
    short v;
    try
    {
      v = n.val;
    }
    catch(NullPointerException e)
    {
      return rescue;
    }
    if( v >= 0 )
      return v;
    else
      return (short) -v;
  }

}
class C2 extends C
{
  C2()
  {
    rescue = 4;
  }



  public short run()
  {
    return abs((new Num[3])[1]);
  }









}
```

```
package p;

class Num { short val; }

public abstract class C
{
  protected short rescue;
  public C() {
     0: load r 0
     1: invokedefinite Object()
     2: return
  }
  public short abs(Num n) {
     0: load r 1
     1: getfield Num.val
     2: store s 2
     3: goto 7
     4: pop 1
     5: getfield this p.C.rescue
     6: return s
     7: load s 2
     8: if lt s 0 goto 11
     9: load s 2
    10: return s
    11: load s 2
    12: numop s neg
    13: return s
    0-2: NullPointerException => 4
  }
}
class C2 extends C
{
  C2() {
     0: load r 0
     1: invokedefinite p.C()
     2: push s 4
     3: putfield this p.C.rescue
     4: return
  }
  public short run() {
     0: load r 0
     1: push s 3
     2: new p.Num[]
     3: push s 1
     4: arrayload r
     5: invokevirtual p.C2.abs(p.Num)
     6: return s
  }
}
```

Figure 1: Java Card program example (left), corresponding JCVML*e* program (right)

References

[FM99] Stephen N. Freund and John C. Mitchell. A formal framework for the Java bytecode language and verifier. In Loren Meissner, editor, *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 147–166, N. Y., November 1–5 1999. ACM Press.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* The Java Series. Addison-Wesley, Boston, Mass., 2000. See also http://java.sun.com/docs/books/jls/index.html.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, MA, USA, second edition, 1999. See also http://java.sun.com/docs/books/vmspec/index.html.

[Mar00] Renaud Marlet. Proposition of hierachy of languages to be studied in SecSafe. Technical Report SECSAFE-TL-004, v1.0, Trusted Logic, December 2000.

[MLM00] Renaud Marlet and Daniel Le Métayer. First note on industrial needs in terms of language, API and security properties of Java Card. Technical Report SECSAFE-TL-002, v1.2, Trusted Logic, November 2000.

[Sun99] Sun Microsystems. Java Card 2.1 Runtime Environement (JCRE) Specification, June 1999. See http://java.sun.com/products/javacard/javacard21.html.

[Sun00] Sun Microsystems. Java Card 2.1.1 Virtual Machine Specification, May 2000. See http://java.sun.com/products/javacard/javacard21.html.

A Brief Comparison With Freund and Mitchell's Formalization

Freund and Mitchell (F&M) have proposed a formalization of the Java bytecode language [FM99]. Their published formalization describes the language JVML$_f$, that only includes a fraction of the JCVML, consisting of representative instructions.

Moreover, the authors have developed a prototype implementation of a bytecode verifier[3] based on this formalization, which is able to verify "a large fraction of the JDK library". Because of the frequency of most missing instructions in JVML$_f$, we believe that that the prototype supports many more instructions than the given formalization.

In fact, the prototype makes use of a translator that is dedicaded to verification and that maps Java bytecode to a subset of JVML$_f$. The authors say that "this translation preserves the structure and the data flow of the original program, but utilizes a small core verifier to perform type synthesis." For instance, it replaces `invokevirtual` by a few `pops` (for the arguments) and a `push` (for the result). The reason is that their bytecode verification is only interested in the effect on the operand stack; it is thus enough to analyze each method in turn. As can be seen from this example, the data flow that is actually preserved is just the intraprocedural dataflow. This abstraction is not suitable for interprocedural analyses, as will be needed in SecSafe, e.g., to trace the flow of private data. (See also the end of section 8.4 of [FM99], where the authors say in particular "we have not included interprocedural analysis or global information in our framework".)

We should thus be cautious about relying on, or comparing to [FM99], as the authors do not have the same goals as ours. Their formalization may have abstracted away information that is useful to us. Our framework is not fundamentally different from the formalization in [FM99] though. Differences are listed below.

Program Structure. The program structure described in section 2 does not conceptually differ much from Freund and Mitchell's descriptors and environment. The differences are the following.

- F&M use names (and types) as well as environment functions mapping names (and types) to data structures holding information. Our framework provides direct access to data structure and information without having to manipulate names, except for purely dynamic items, i.e., virtual and interface method identifiers.
- F&M use heterogeneous tuples whereas our framework gives names to each component of a structured data (as in a record).
- Our program structure provides more information than F&M's. E.g., given a class, one gets access to its subclasses, methods, package.

Instructions. In general, JCVML*e* instructions have access to more information than their counterpart in JVML$_f$. E.g., the `load` instruction knows the type of the value to load. Moreover, the following instructions are missing from [FM99].

- `dup`, `swap`: these instructions are not in the formalization but probably are in the front-end translator of the verifier as they are quite common.
- `getstatic`, `putstatic`: these instructions model global variable. We suppose that they have been excluded from [FM99] because, as the authors' concern is intraprocedural, these instructions are uninteresting: they behave exactly as `load` and `store`. This is not the case for interprocedural analyses.

[3] It maybe only is a type checker.

- `lookupswitch`, `tableswitch`: these branching instructions are more complex than `if` but are not fundamentally different regarding the flow of control and data. Although they could be expressed using a bunch of `if`, it seems more suitable to have a "native" treatment of these swtich instructions. There is no semantic difficulty.
- `inc`: this instruction is not fundamental as it can be easily and faithfully expressed using `load`, `push`, `numop` and `store`. We have kept it in our formalization just for the sake of exhaustivity. However, it should not be too much of a burden in the context of our framework: it is the only instruction that can trivially be rewritten in terms of others.

The following instructions have a different name or are more general in JCVML*e*:

- `numop` encompasses `add`. There is no technical difference regarding analysis, except that that some operators, as specified by a parameter to `numop`, are unary rather than binary.
- `if` encompasses `ifeq`. Again, there is no fundamental difference besides the varying number of operands. The arguments of `if` (see above) just allow all conditional branching instructions to be modeled with a single one.
- `new` encompasses both `new` and `newarray` of [FM99]: the type argument is made richer to include not only class types but also array types.
- `return` encompasses `returnvalue`. A result type parameter says if the instruction returns `void` or a value.
- `invokedetermined` factorizes `invokestatic` (missing in [FM99]) and `invokespecial`, expressing the fact that both instructions actually invoke a method that can statically be determined at link time.

B Linking Issues

Linking in the JCVM is different from linking in the JVM. We understand here *linking* as anything that allows two program units to run jointly, i.e., that allows a program unit to refer to another one. (Linking also occurs internally, in a given program unit, although it is not as general as between two program units.) Program items involved in linking are packages, classes, interfaces, methods and fields. We call them *linkable program items*. Linking data is used both to specify the class structure of a program and to annotate instructions with information used at execution time.

Names vs. Tokens. Linking data in the JCVM is different from linking data in the JVM.

- In the JVM, linking is based on *names* (UTF-8 strings) and types. More precisely, packages, classes and interfaces are identified by their (fully-qualified) name, fields are identified by their class, name and type[4], methods are identified with their class and signature (name and argument types). Some implementations also use *indices* to identify fields and methods internally, especially at runtime. Indices are used, e.g., to get access to a virtual method in the method table of a class.
- In the JCVM, linking is based on *tokens* and *offsets*. Tokens are small numbers identifying externally visible (i.e., public or protected) program items; tokens are also used as indexes into various tables in the CAP file as well as runtime data structures. Linkable program items that are not externally visible (i.e., private or package-visible) are referred to using direct offsets into CAP file components rather than tokens. The mapping from names to tokens and offsets is performed during conversion from class files to CAP files [Sun00]. In addition to tokens, JCVM packages are also referenced using AIDs (application identifiers made of byte sequences).

As described in this document, all static linking information in the JCVM*e* is expressed using direct references to the program structure, i.e., elements of the domains *Package*, *Class*, *Interface*, *Field* and *Method*. Dynamic linking information is abstracted into opaque IDs of fields and methods. Low-level details of the JCVM are hidden and the JCVM*e* is made closer to the JVM.

Note that names should *never* be used when expressing the program semantics except to get access to a class or interface given its fully qualified name, e.g., to express the semantics of an API method. Field and method IDs should *always* be used instead. One of the major reason is that names are generally not available in the JCVM[5].

References to Packages.

- In the JVM, there are no explicit references to packages. Packages only occur as prefixes in class and interface names.
- In the JCVM, references to package are based on tokens as well as AIDs.

In the JCVM*e*, packages are made explicit in the program structure: programs contain packages, that in turn contain classes and interfaces. Besides, packages also provide access to their AID.

References to Classes and Interfaces. There are three kinds of references to classes and interfaces: those that are used for building the class structure, those that are used to designate fields and methods,

[4] According to Java semantics, two fields in the same class cannot have the same name and different types. Assuming the code is well typed, the field name and class are enough to identify a field.

[5] CAP files do not contain any name. Only Export files contains some names, corresponding to program items that are externally visible (i.e., public or protected). Most names can be made available though via the (optional) Debug Component of the CAP file.

and those that are used to build reference types. These reference types occur in field and methods descriptors as well as parameters of some instructions.

- In the JVM, all references to classes and interfaces are based on names.
- In the JCVM, references to classes and interfaces are based either on offsets or on package and class tokens.

In the JCVM*e*, all references to classes and interfaces provide a direct accesses to the representation of the program structure, i.e., elements of the *Class* and *Interface* domains.

References to Fields and Methods. There are two kinds of references to field and methods: those that are used for building the class structure, and those that are used by the instructions that access fields or invoke methods. References used for class structuring are as follows.

- In the JVM, references to fields and methods that are relevant for building the class structure are provided by the nesting of data structures in the program files.
- This is also the case of the JCVM Descriptor Component, but not of the other CAP file components, that refer to each other using tokens and offsets[6].

In the JCVM*e*, field and methods references that represent program structure translate into direct access to the corresponding high-level data structure, i.e., elements of the *Field* and *Method* domains. Only the references that are used by the instructions that invoke virtual and interface methods have to be explicitly retained. For homogeneity reasons, references for accessing fields and invoking definite methods are also kept. This is further detailed in the following paragraphs.

Field and Method IDs. As explained below, fields and methods are given *IDs* that are used for identification in various lookups required by the instruction semantics, e.g., identification of actual method to call when invoking a virtual method. (We use term "ID" rather than "identifier" to stress that, although it is conceptually used as a name, ID equality is *not* as name equality.) Field and method IDs have the following properties:

- Two field IDs or static method IDs are equal iff the corresponding fields or methods are equal.
- Two virtual method IDs are equal iff they correspond to methods that are either equal or such that one overrides the other one.
- An interface method ID is equal to a virtual method ID iff the class of the corresponding virtual method implements the corresponding interface and if the corresponding methods match according to the Java semantics.
- Two interface method IDs are equal iff the corresponding methods are equal.

Also note that ID domains for class fields, instance fields, class methods and instance methods are all disjoint.

Static Field Access.

- In the JVM, instructions that access a static field refer to a class name, a field name and a field type.
- In the JCVM, instructions that access a static field refer either to an offset into the *static field image* (as specified by the *Static Field Component* [Sun00]), or to tokens representing a package, a class and a field, which in turn refer to an offset into the static field image.

[6]The Descriptor Component is not needed for JCVM execution. It is used to provide enough typing information to allow bytecode verification. We assume that the Descriptor Component is always present.

In the JCVM*e*, instructions that access a static field directly refer to a static field in the *Field* domain (see §2.6). Each static field has a unique *static field ID* in the *FieldID* domain that can be tested for equality, e.g., when looking up a field in a table.

Position information that is useful at runtime (offset into the static field image and position into the class field value area) is not kept. It is not a problem because it is not needed to express the operational semantics.

Instance Field Access.

- In the JVM, instructions that access an instance field refer to a class name, a field name and a field type.
- In the JCVM, instructions that access an instance field refer to a field token and a class.

In the JCVM*e*, instructions that access an instance field directly refer to an instance field in the *Field* domain (see §2.6). Each instance field has a unique *instance field ID* in the *FieldID* domain that can be tested for equality, e.g., when looking up a field in a table.

Position information that is useful at runtime (field token and position into the class field value area) is not kept. It is not a problem because it is not needed to express the operational semantics.

Definite Method Invocation. *Definite methods* consist of class methods (also called static methods), superclass and private instance methods, as well as instance initialization methods.

- In the JVM, instructions for definite method invocation, i.e., instructions `invokestatic` and `invokespecial`, refer to a definite method via a class name and a method signature.
- In the JCVM, instructions for definite method invocation refer to a definite method, either directly via an offset if the method is defined in this package, or indirectly via tokens representing a package, a class and a field if the method is defined in another package.

In the JCVM*e*, the instruction for definite method invocation directly refers to a definite method in the *Method* domain (see §2.5). Each definite method has a unique *definite method ID* in the *MethodID* domain that can be tested for equality, e.g., when looking up a method in a table.

Virtual Method Invocation.

- In the JVM, the instruction for virtual method invocation refers to an instance method via a class name and a method signature. It does not mean that the corresponding method in this class should be called. It only defines a base instance method that is possibly overridden in subclasses. In JVM implementations, an index is often associated to virtual methods to rapidly access, at run time, the actual method to call given the method table of the class of the runtime object. Alternatively, the class hierarchy has to be walked to find the appropriate method.
- In the JCVM, the instruction for virtual method invocation refers to an instance method via a class reference and a virtual method token. As in the case for the JVM, this only defines a base instance method that is possibly overridden in subclasses. As in the JVM case, the virtual method token somehow represents the index of the method in the virtual method table of the class. However, this table does not exist as such, it is fragmented along the class hierarchy, both inside and across CAP files. In JCVM implementations, the *Class Components* of the CAP files are traversed at runtime to determine the actual method to invoke. Because of the CAP file format, determining the right method to invoke involves using tokens as indices into various tables and doing (simple) arithmetic computations.

In the JCVM*e*, the instruction for virtual method invocation directly refers to a virtual method in the *Method* domain (see §2.5). As above, this method actually is a base method that is possibly overridden in subclasses. The actual method to call, given the class of a runtime object, is in the first class defining this base method (possibly overriding it) walking up the class hierarchy from the runtime object class towards the class of the base method. To that effect, each virtual method is given a *virtual method ID* that can be compared with other virtual method IDs in the class hierarchy. As explained above (see also §2.5), any overriding method has the same identifier as the method it overrides.

Interface Method Invocation.

- In the JVM, the instruction for interface method invocation refers to a virtual method via an interface name and the signature of a method defined in this interface. At run time, when the class of the referenced object is known, one possible implementation (as described in the JVM specification [LY99]) is to search its method table for a method that matches the method signature. Other implementations can be more efficient, e.g., replacing the search by an indexed table lookup.
- In the JCVM, the instruction for interface method invocation refers to an instance method via a interface reference and an interface method token. At run time, when the class of the referenced object is known, the *implemented interface table* of this class, and possibly of all its superclasses, is searched to find an entry for the interface referenced by the instruction. This entry includes an indexed table to map the interface method token into a virtual method token. This virtual method token is then to be used as if performing a virtual call on the referenced object, as described in the previous paragraph.

In the JCVM*e*, the instruction for interface method invocation directly refers to an interface method in the *Method* domain (see §2.5). As in the virtual method invocation case, this is just an indication of a base method. The actual method to call, given the class of a runtime object, is the first class defining this base method walking up the class hierarchy from the runtime class towards the class of the base method. To that effect, each interface method is given an *interface method ID* that can be matched against virtual method IDs found in the class hierarchy (see §2.5).