Operational Semantics of the Java Card Virtual Machine

Igor A. Siveroni

Imperial College London. U.K.

Abstract

We present the operational semantics of Carmel, a language that models the Java Card Virtual Machine Language. We use the *instruction set* and the *program structures* proposed in [1]. We define a small-step relation between program configurations, including rules for exception handling, arrays and subroutines. We also include the basic structures needed to model object ownership and the Java Card firewall.

Author:	Igor Siveroni
Date:	October, 2002
Number:	SECSAFE-ICSTM-010
Version:	1.0
Classification:	Confidential
Status:	Submitted for publication.
	Journal version of Secsafe-ICSTM-001

Key words: Java Card, Semantics
PACS:

1 Introduction

Java Card was first introduced in 1996 as a new technology that enables Java applets to be loaded and executed in new generation smart cards and other devices with limited memory. Java Card is a simplified version of Java

Email address: `siveroni@doc.ic.ac.uk` (Igor A. Siveroni).

- it does not include threads, floating point numbers and multi-dimensional arrays - and has a relatively small size API. Java Card applications are also small due to the memory limitations of the Smart Card processor. All these factors combined with the the security-critical nature of its applications have made it an ideal case-study for the application of formal methods.

The Java Card platform [2] is defined by three (natural language) specifications: The *Java Card Virtual Machine* (JCVM) *Specification* [3], the *Java Card Application Programming Interface* (API) [4], and the *Java Card Runtime Environment* (JCRE) *Specification*[5]. The specifications of other important components of the Java Card system - such as the bytecode verifier, applet installer and CAP converter - can either be found scattered in the specification of the main components or deduced from their counterparts in the Java system (e.g. bytecode verification [6]). These specifications are sometimes ambiguous and contradictory, and lack mathematical rigour. This works tackles the first problem: to provide a formal specification of the JCVM.

We present the operational semantics of Carmel, a language that models the Java Card Virtual Machine Language. We use the *instruction set* and the *program structures* proposed in [1]. We define a small-step relation between program configurations, including rules for exception handling and subroutines. We also include the basic structures needed to model object ownership and the Java Card firewall.

One of the goals of the operational semantics is to provide an adequate framework for the formalisation of program (e.g. flow) analyses for the JCVM[7]. Further refinements to the semantics [8] will be directed - among other things - towards this goal.

1.1 Notation

A **Domain** is defined as a set equipped with functions. For convenience, in some cases we define new domains as records. A record defined by

$$\mathsf{Dom} = (f_1 : \mathsf{Dom}_1) \times ... \times (f_n : \mathsf{Dom}_n)$$

is equivalent to the domain Dom equipped with the interface $f_i \in \mathsf{Dom} \to \mathsf{Dom}_i$, for $0 < i \leq n$. Given $e \in \mathsf{Dom}$, access to element f_i is written as $e.f_i$ and element update is performed by $e[f_i \mapsto v]$, where $v \in \mathsf{Dom}_i$. To simplify notation, we will usually write $o.f(args)$ instead of $f(o, args)$.

We write $[X]$ to denote the set of finite arrays of elements of domain X. The length of an array can be known using *length* : $[X] \to \mathsf{int}$. The indices of an array x range from 0 to $x.length - 1$. Elements of the array x can be accessed using the standard $x[i]$ notation. The i-th element of an array is updated by $x[i \rightarrow v]$.

2 Carmel Syntax and Program Structure

2.1 Program Structures

Java Card programs, unlike Java, are not represented as `class` files. Instead, Java Card CAP - converted applet - files are used. A Java Card CAP file is the binary representation of a package of classes that can be installed on a device and used to execute the package's classes on a Java Card virtual machine[3]. In our framework, the information contained in the loaded CAP files is represented by a *program structure*. A program P contains a set of packages and each package a set of classes and interfaces. A package is uniquely identified by an AID (application identifier). We define the following domains:

$$P \in \mathsf{Program}\ p \in \mathsf{Package}\ aid \in \mathsf{AID}\ cl \in \mathsf{Class}\ iface \in \mathsf{Interface}$$

Classes and packages can be extracted from a program using the interface defined in [1]. Similarly, methods and fields can be obtained from classes and interfaces. We will only point out the elements relevant to this presentation and that require some explanation. Classes and interfaces hold information about direct fields and methods, inherited elements have to be looked up in the superclass. A set of auxiliary functions is defined to help extract information from the class hierarchy. The functions $\mathit{superClasses}(cl)$ and $\mathit{superInterfaces}(iface)$ return the set of all the superclasses and superinterfaces of class cl and interface $iface$, respectively. We write $cl \in \mathit{implements}(iface)$ if cl implements interface $iface$.

Names are dropped from CAP files. All references to classes, interfaces, methods and fields have been resolved to their respective representation structure. However, in order to make the presentation more readable, names - concrete syntax - are introduced when necessary. Thus, when we write `Object` we are actually referring to the structure associated to the `java.lang.Object`[1] class in the program.

A method structure m contains information about a method's ID ($m.id$),used for virtual method invocation (section 6.5), its type ($m.type$) and the class or

[1] By default we'll assume that classes without prefix belong to the `java.lang` package

interface ($m.classI$) where it was declared. We write $|m|$ to denote the method's arity. A non-abstract method has additional information concerning its body. The sequence of instructions that make up a method's body is accessed via addresses ($\mathsf{Address}$). We use $m.instructionAt(addr)$ to access the instruction at address $addr$ in method m. The first instruction of a method is located at address 0 and the instruction following the instruction at address $addr$ can be found at $addr + 1$.

A field is uniquely identified by a field id. A field structure f contains information about the field id ($f.id$) and type ($f.type$).

A JCVML program deals with two kinds of data types: primitive types and references (Figure 1). The primitive types supported by Java Card are `boolean`, `byte` and `short`. Some implementations may also support the `int` type. References are either class instances or one-dimensional arrays. A method type τ_{m} is made of two components: the sequence of types that represent the types of the parameters (it could be ϵ), and the type of the result. The notation $(\tau_i)_1^n \to \tau_{\mathtt{rm}}$ is used to extract the two components. For clarity, we will use this syntactic representation of types instead of the structures defined in [1]. The subtype relation $\tau \preceq \tau'$ (Figure 1)is defined following the "can be assigned to" criteria as indicated by the description of `checkcast` and `instanceof` in [5].

The Carmel instruction set can be divided into six groups, each dealing with a different aspect of the language: imperative core (C), objects (O), method invocation (M), arrays (A), exceptions (E) and subroutines (S). The instruction set[2] is defined in Figure 1. As JCVM instructions, most Carmel instructions are typed and operate on values that have an expected runtime type - the operand type. One of the main differences between the JCVM language and Carmel resides in the way the operand type information is encoded. In Carmel, instead of having a specialised instruction for each operand type (e.g. `astore`, `istore`), instructions take the operand type as an argument (e.g. `store a 2`). This simplification reduces the number of instructions.

As mentioned above, references to methods, fields and classes have already been resolved and, therefore, can be accessed directly from wherever they are referenced. Thus, the field structure f can be extracted directly from the `getstatic` f instruction. Constants c are introduced by the `push` instruction and can be either integers or the special constant `null`. Numeric operators (`add`,`sub`,etc.) and relational operators (`eq`,`ne`) are denoted by op and cmp. Number of words and array indexes are denoted by n and d, and i, respectively, while $addr$ is used to denote an instruction address.

[2] For this presentation, we have removed the `keyswitch`, `indexswitch` instructions, and those that take the `this` keyword as argument

$$\begin{array}{rcl}
\tau \in \mathsf{Type} & ::= & \mathsf{ReferenceType} \mid \mathsf{PrimitiveType} \\
\mathsf{PrimitiveType} & ::= & \mathtt{boolean} \mid \mathtt{byte} \mid \mathtt{short} \mid \mathtt{int} \\
\tau_{\mathtt{r}} \in \mathsf{ReferenceType} & ::= & \mathsf{ArrayType} \mid \mathsf{SimpleRef} \\
\mathsf{SimpleRef} & ::= & \mathsf{ClassName} \mid \mathsf{InterfaceName} \\
\mathsf{CompType} & ::= & \mathsf{SimpleRef} \mid \mathsf{PrimType} \\
\mathsf{ArrayType} & ::= & \mathsf{CompType}\mathtt{[\ ]} \\
\tau_{\mathtt{rm}} \in \mathsf{ResultType} & ::= & \mathsf{Type} \mid \mathtt{void} \\
\mathsf{MethodType} & ::= & \mathsf{Type}^{*} \rightarrow \mathsf{ResultType}
\end{array}$$

$$\frac{\tau \in \mathsf{ComponentType}}{\tau \preceq \tau} \qquad \frac{cl' \in \mathit{superClasses}(cl)}{cl \preceq cl'} \qquad \frac{\mathit{iface} \in \mathit{implements}(cl)}{cl \preceq \mathit{iface}}$$

$$\frac{}{\mathit{iface} \preceq \mathtt{Object}} \qquad \frac{\mathit{iface}' \in \mathit{superInterfaces}(\mathit{iface})}{\mathit{iface} \preceq \mathit{iface}'}$$

$$\frac{}{\tau\mathtt{[\ ]} \preceq \mathtt{Object}} \qquad \frac{\tau \preceq \tau'}{\tau\mathtt{[\ ]} \preceq \tau'\mathtt{[\ ]}} \qquad \frac{\mathit{iface} \in \mathit{implements}(\mathtt{Array})}{\tau\mathtt{[\ ]} \preceq \mathit{iface}}$$

$$\begin{array}{rcll}
I \in \mathsf{Instruction} & ::= & \mathtt{nop} \mid \mathtt{push}\ t\ c \mid \mathtt{pop}\ n \mid \mathtt{swap}\ n_1\ n_2 \mid \mathtt{dup}\ n\ d & \text{(C)} \\
 & & \mathtt{numop}\ t\ \mathit{op}\ t \mid \mathtt{goto}\ \mathit{addr} \mid \mathtt{if}\ t\ \mathit{cmp}\ \mathtt{goto}\ \mathit{addr} & \\
 & & \mathtt{load}\ t\ i \mid \mathtt{store}\ t\ i \mid \mathtt{inc}\ t\ i\ c & \\
 & & \mathtt{new} \mid \mathtt{checkcast}\ \tau \mid \mathtt{instanceof}\ \tau & \text{(O)} \\
 & & \mathtt{getstatic}\ f \mid \mathtt{putstatic}\ f \mid \mathtt{getfield}\ f \mid \mathtt{putfield}\ f & \\
 & & \mathtt{invokevirtual}\ m \mid \mathtt{invokedefinite}\ m & \text{(M)} \\
 & & \mathtt{invokeinterface}\ m \mid \mathtt{return}\ t \mid \mathtt{return} & \\
 & & \mathtt{arraylength} \mid \mathtt{arrayload}\ t \mid \mathtt{arraystore}\ t & \text{(A)} \\
 & & \mathtt{throw} \mid \mathtt{jsr}\ \mathit{addr} \mid \mathtt{ret}\ i & \text{(E) \& (S)} \\
t \in \mathsf{OpType} & = & \{\mathtt{b}, \mathtt{s}, \mathtt{i}, \mathtt{r}\} &
\end{array}$$

Fig. 1. Carmel Types, Subtyping Rules and Instruction Set

3 Runtime Structures and Values

3.1 Objects and the Heap

Objects are stored in the heap. The heap is defined as a mapping from locations to objects, which can be class instances or arrays.

$$\begin{aligned} H \in \mathsf{Heap} &= \mathsf{Location} \rightarrow \mathsf{Object} \\ \mathsf{Object} &= \mathsf{ClassInst} \cup \mathsf{ArrayObject} \end{aligned}$$

A class instance is defined by:

$$\begin{aligned} o \in \mathsf{ClassInst} &= (\mathit{class} : \mathsf{Class}) \times (\mathit{owner} : \mathsf{Owner}) \times \\ &\quad (\mathit{entryPoint} : \mathsf{EntryPoint}) \times (\mathit{fieldValue} : \mathsf{FieldID} \rightarrow \mathsf{Value}) \\ \mathsf{Owner} &= (\mathit{context} : \mathsf{AID} \cup \{JCRE\}) \times (\mathit{aid} : \mathsf{AID} \cup \{JCRE\}) \\ \mathsf{EntryPoint} &= \{\mathtt{perm}, \mathtt{temp}, \mathtt{no}\} \end{aligned}$$

The *class* element stores the object's class. *fieldValue* is a map from field ids to runtime values. *owner* contains information about the owning context (owner's package AID) and applet, while *entryPoint* indicates if the object is a permanent or temporary JCRE entry point.

New objects are created and allocated by the function *newObject* :

$$\begin{aligned} & \mathit{newObject}(\mathit{own}, \tau, \mathit{ep}, H) = (H', \mathit{loc}) \Leftrightarrow \\ & \quad \mathit{loc} \notin \mathit{dom}(H) \wedge H' = H[\mathit{loc} \mapsto o] \wedge o \in \mathsf{ClassInst} \\ & \quad o.\mathit{class} = \tau \wedge o.\mathit{owner} = \mathit{own} \wedge o.\mathit{entryPoint} = \mathit{ep} \\ & \quad o.\mathit{fieldValue} = \{(f.\mathit{id}, \mathit{def}(\tau)) \mid f \in \mathit{instanceFields}(\tau)\} \end{aligned} \tag{1}$$

where *instanceFields* is the set of non-static fields of a class, including the fields defined in its superclasses, and $\mathit{def}(\tau)$ returns the default value for a particular type τ (numeric types map to 0 and reference types to `null`). For this presentation we assume that fields and array values are initialised to the default values indicated by their types.

For security reasons, sometimes it is not desirable to preserve data stored in objects across sessions. An object is transient if the data stored in its fields is cleared whenever the card is reset or the applet is deselected. Currently only arrays with primitive components or references to `Object` can be designated as transient objects. There are two types of transient objects, namely, `CLEAR_ON_RESET` and `CLEAR_ON_DESELECT`. The interaction between transient objects and the applet firewall is explained in section 5. An array object is defined by:

$$\begin{aligned}
a \in \mathsf{ArrayObject} = {} & (\mathit{eleType} : \mathsf{Type}) \times (\mathit{length} : \mathsf{integer}) \times \\
& (\mathit{owner} : \mathsf{Owner}) \times (\mathit{global} : \mathsf{boolean}) \times \\
& (\mathit{transient} : \mathsf{Transient}) \times (\mathit{values} : [\mathsf{Value}]) \\
\mathsf{Transient} = {} & \{\mathtt{CLEAR\_ON\_RESET}, \mathtt{CLEAR\_ON\_DESELECT}, \mathtt{NOT\_TRANSIENT}\}
\end{aligned}$$

New arrays are created and allocated with the *newArray* function:

$$\begin{aligned}
& \mathit{newArray}(\mathit{own}, \tau, n, g, t, H) = (H', \mathit{loc}) \Leftrightarrow \\
& \quad \mathit{loc} \notin \mathit{dom}(H) \wedge H' = H[\mathit{loc} \mapsto a] \wedge a \in \mathsf{ArrayObject} \\
& \quad a.\mathit{eleType} = \tau \wedge a.\mathit{length} = n \wedge a.\mathit{owner} = \mathit{own} \\
& \quad a.\mathit{global} = g \wedge a.\mathit{transient} = t \wedge a.\mathit{values}.\mathit{length} = n \\
& \quad \forall i.\ 0 \leq i < n.\ a.\mathit{values}[i] = \mathit{def}(\tau)
\end{aligned} \tag{2}$$

3.1.1 Heap Update

Class instances and array objects are identified by their location in the heap. Similarly an element location, defined by

$$\mathit{eloc} \in \mathsf{ElementLocation} = (\mathsf{Location} \times \mathsf{FieldID}) + (\mathsf{Location} + \mathsf{ShortValue}),$$

is used to identify instance fields and array entries of class instances and array objects, respectively. Thus, we define a more succinct definition of field and array update with the *updateElement* function:

$$\begin{aligned}
& \mathit{updateEle} : \mathsf{Heap} \times \mathsf{ElemetLocation} \times \mathsf{Value} \rightarrow \mathsf{Heap} \\
& \quad \mathit{updateEle}(H, (\mathit{loc}, \mathit{id}), v) = H[\mathit{loc} \mapsto o'], \\
& \qquad\qquad o' = H(\mathit{loc}).\mathit{fieldValues}[\mathit{id} \mapsto v] \\
& \quad \mathit{updateEle}(H, (\mathit{loc}, i), v) = H[\mathit{loc} \mapsto a'], \\
& \qquad\qquad a' = H(\mathit{loc}).\mathit{values}[i \mapsto v]
\end{aligned} \tag{3}$$

Given array a, we will sometimes write $a[i]$ instead of $a.\mathit{values}[i]$.

3.2 Values and Runtime Data Types

The JCVM supports two kind of values: *primitive values* and *reference values*. Primitive values consist on numeric values (`byte`, `short` and `int`) and return addresses. Reference values consist of references to class instances and arrays (locations, section 3.1) and the special constant `null`. Java Card virtual machine values can be:

$$\begin{aligned}
v \in \mathsf{Value} &= \mathsf{PrimitiveValue} \;\cup\; \mathsf{ReferenceValue} \\
\mathsf{PrimitiveValue} &= \mathsf{NumericValue} \;\cup\; \mathsf{ReturnAddressValue} \\
\mathsf{NumericValue} &= \mathsf{ByteValue} \;\cup\; \mathsf{ShortValue} \;\cup\; \mathsf{IntegerValue} \\
\mathsf{ReferenceValue} &= \mathsf{Location} \cup \{\texttt{null}\}
\end{aligned}$$

$\mathsf{ReturnAddressValue}$ values are runtime representations of $\mathsf{Address}$ elements. Storage is managed by the JCVM in terms of an abstract storage unit called *word*. A word is large enough to hold reference and numeric values, with the exception of integers. Two words are large enough to hold a value of type `i`. The function *nbWords* returns the number of words of a value or a sequence of values while *nbBytes* returns the number of bytes of a value or element location:

Therefore, the Java types `byte` and `short` - and in some implementations, the `int` type - are supported by the JCVM. Values of the Java type `bool` are implemented as byte values, where 1 represents `true` and 0 `false`.

Operations between numeric values are performed using the functions *applyUnary* and *applyBinary*:

$$\begin{aligned}
\mathit{applyUnary} &: \mathsf{UnaryNumericOperator} \times \mathsf{Value} \times \mathsf{OpType} \times \mathsf{OpType} \\
\mathit{applyBinary} &: \mathsf{BinaryNumericOperator} \times \mathsf{Value} \times \mathsf{Value} \times \mathsf{OpType}
\end{aligned}$$

The *applyUnary* function requires as arguments the operand and result types. The result type is needed by the type conversion operator `to`. Binary operations do not perform type conversions and, therefore, operand and result types are the same.

`bool` and `byte` values have to be sign-extended before they are pushed to the operand stack. Similarly, `short` values placed on the stack have to be truncated before they are stored in to a field or array element of type `bool` or `byte`. Value conversions from byte to short, and from short to byte are performed by the *toShort* and *toByte* functions, respectively.

$$\frac{H(loc) \in \mathsf{ArrayObject} \quad \tau = H(loc).eleType}{H \vdash loc \ : \ \tau\texttt{[ ]}} \qquad \frac{H(loc) \in \mathsf{ClassInst} \quad \tau.class = H(loc).class}{H \vdash loc \ : \ \tau}$$

$$\frac{\tau \in \mathsf{ReferenceType}}{H \vdash \texttt{null} \ :\preceq \ \tau} \qquad \frac{(H \vdash v \ : \ \tau') \wedge (\tau' \preceq \tau)}{H \vdash loc \ :\preceq \ \tau}$$

Fig. 2. Runtime Typing

Figure 2 details the rules for the runtime typing judgements $H \vdash v \ : \ \tau$ and $H \vdash v \ :\preceq \ \tau$. The judgement $H \vdash v \ : \ \tau$ assigns the Java reference type τ to the location *loc*. The judgement $H \vdash v \ :\preceq \ \tau$ is used to indicate that runtime value v can by assigned to a Java type τ. In the general case, $H \vdash v \ :\preceq \ \tau$ is true if the runtime type of v is a subtype of τ. In the case that v is a short runtime value the relation is more flexible and allows τ to be any one-word numeric type or boolean. This is in response to the fact that one-word numeric values are sign-extended to values of type short whenever they are pushed on to the stack. The special null reference can be assigned to any reference type.

3.3 Frames

A frame contains information about the state of execution of a method. It is defined as follows:

$$\begin{aligned} F \in \mathsf{Frame} &= \mathsf{Owner} \times \mathsf{Method} \times \mathsf{Address} \times \mathsf{LocalVar} \times \mathsf{OperandStack} \\ \mathsf{StackValue} &= \mathsf{Value} - \mathsf{ByteValue} \\ S = v_1{::}\ldots{::}v_n \in \mathsf{OperandStack} &= \mathsf{StackValue}^* \\ V \in \mathsf{LocalVariable} &= [\mathsf{StackeValue}] \end{aligned}$$

A frame $F = \langle own, m, pc, V, S \rangle$ contains the following components:

- The owning context and applet, *own.context* and *own.aid*, respectively.
- The current method structure $m \in \mathsf{Method}$.
- The program counter register $pc \in \mathsf{Address}$, which contains the address of the instruction currently being executed in method m.
- The array of local variables V.
- The operand stack S.

JCVM instructions take their dynamic operands from the operand stack and, in some cases, the array of local variables. The operand stack is defined as a sequence of values $S = v_1{::}\ldots{::}\ldots v_n$ where values on top of the stack appear on the right-hand side of the sequence i.e. v_n is on top of the stack above. The empty stack is represented with the symbol ϵ. An element can be added to a stack with the **::** operator (S::v) and two stacks can be concatenated with the **:** operator (S:S'). For clarity we will write $\mathrm{v}_1{::}\mathrm{v}_2$ instead of $\epsilon{::}\mathrm{v}_1{::}\mathrm{v}_2$.

The operand stack stores values of all types except those of type `byte` (and `boolean`). As it was mentioned in section 3.2, some operations may require values to be truncated (or sign-extended) after they are popped from (or before they are pushed to) the stack. For such cases, the $\mathit{fromStack}(v, \tau)$ and $\mathit{toStack}(v, \tau)$ functions are used to determine if - by inspecting the type τ - type conversion is required and, if so, call the appropriate type conversion function i.e. *toShort* and *toByte*.

Stack operations do not carry type information, they only require that the integrity of the values is maintained i.e. multiword values must not be split apart.

A local variable is not referenced by name but by the position it occupies in the array of local variables. Unitialised local variables are denoted by $\perp$. The array of local variables can store values of any of the stack types. Local variable slots are not statically typed so values of different types can be stored in the same slot at different times during the execution of a method. Special care must be taken when using multiword values. An integer value stored in $V[i]$ takes two slots and access to $V[i+1]$ is prohibited. We can model this by making $V[i+1] = \perp$ if position i holds an integer value.

We write $V = v_0{::}\ldots{::}v_n$ when local variable array V is initialised with the sequence of values (usually from the operand stack) $v_0{::}\ldots{::}v_n$. Note that $V[0] = v_0$ but v_i does not necessarily correspond to $V[i]$ due to multi-word values.

3.4 Configurations

A runtime configuration describes the state of execution of the JCVM. We define three kinds of configurations:

$$C \in \mathsf{Config} \;=\; \mathsf{RConfig} \,\cup\, \mathsf{EConfig} \,\cup\, \mathsf{HConfig}$$

A running configuration - $\mathsf{RConfig}$ - keeps track of the chain of invoked methods by using a stack of call frames. An exception configuration - $\mathsf{EConfig}$ - represents the state of uncaught exceptions or machine errors. The halt configuration is the result of returning from the method that started the execution of the current applet.

A **running configuration** $C = \langle (jcre, K, H), SF \rangle$ contains the following components:

- The JCRE component $jcre \in \mathsf{JCRE}$ used to model the interaction between the JCVM and the JCRE via the API.
- The heap H, a mapping from locations to runtime objects (class and array instances).
- Static fields memory K, a mapping $\mathsf{FieldID} \rightarrow \mathsf{Value}$.
- The call stack SF, where $SF \in \mathsf{Frame}^*$.

The frame of the method currently being executed - the current method - sits on top of the call stack. Given $SF = SF'{::}\langle own, m, pc, V, S \rangle$, the current instruction I can be extracted with $I = m.instructionAt(pc)$. The next instruction can be found at address $pc + 1$ $(m.nextAddress(pc))$.

Exception and halt configurations represent terminal configurations. They are defined by:

$$\langle (jcre, H, K), loc \rangle \in \mathsf{EConfig} = \mathsf{Heap} \times \mathsf{StaticMem} \times \mathsf{Location}$$

$$\langle (jcre, H, K), \mathtt{halt} \rangle \in \mathsf{HConfig} = \mathsf{Heap} \times \mathsf{StaticMem} \times \{\mathtt{halt}\}$$

where the location loc in the exception configuration is a reference to the uncaught exception object.

4 Exceptions

An **exception** is thrown whenever a program violates the semantic constraints of JCVMLe or any other constraint specified by the programmer. All exceptions are extensions of the `Throwable` class and can be thrown explicitly by the programmer using the `throw` instruction.

An exception is said to be caught if an appropriate **exception handler** is found in the current stack of call frames. If an exception is caught, control is transfered to the start address indicated by the exception handler and execution is resumed at that point. An uncaught exception will cause the virtual machine to halt. The process of throwing and catching an exception is defined by the *catchException* function. Let *loc* be a reference to an exception object:

$$
\begin{aligned}
&\mathit{catchException} : \mathsf{Frame}^* \times \mathsf{Heap} \times \mathsf{Location} \rightarrow \mathsf{Frame}^* \cup \mathsf{Location}\\
&\mathit{catchException}(\epsilon, H, loc) = loc\\
&\mathit{catchException}(SF{::}\langle own, m, pc, V, S\rangle, H, loc) =\\
&\qquad\qquad\qquad \begin{cases} \mathit{catchException}(SF, H, loc) & \text{if } pc' = \bot\\ \text{SF}{::}\langle own, m, pc', V, loc\rangle & \text{if } pc' \neq \bot \end{cases}\\
&\quad \text{where } H \vdash loc \ : \ \tau \wedge pc' = \mathit{findHandler}(m, pc, H, loc)
\end{aligned}
$$

The *catchException* function searches the stack of call frames until the first appropriate exception handler is found [3, Chapter 7]. The $\mathit{findHandler}(m, pc, \tau)$ function searches for the first exception handler in method m that can handle an exception of class τ at address *pc*. Information about exception handlers is kept by the *m.exceptionHandlers* structure. The detailed semantics of *findHandler* is described in [9].

4.1 Runtime Exceptions

Runtime exceptions may be raised by the JCVM during the execution of a program. Being memory usage an important concern, the creation of unique runtime exception objects per class is enforced. These are JCRE owned, pre-allocated exceptions objects designated as temporary JCRE entry points. The JCRE keeps a reference to each instance of the Java runtime exception objects in the table *eInst*. We define the following:

$$
\begin{aligned}
\mathsf{ExceptionInstances} &= \mathsf{RTException} \mapsto \mathsf{Location}\\
\mathit{eInst} &\in \mathsf{JCRE} \times \mathsf{RTException}\\
\mathit{getException} &\in \mathsf{JCRE} \times \mathsf{RTException} \ \rightarrow \ \mathsf{Location}
\end{aligned}
$$

eInst gives us access to a function that maps Java Card runtime exception classes to the locations of the respective (unique) exception object.

A mapping *eInst* is well-formed with respect to heap H - we write $H \models \mathit{eInst}$ -if each runtime exception class kept in *eInst* maps to the location of a JCRE owned instance of the same class stored in H. All instances must also be JCRE temporary entry points:

$$\begin{aligned}
&\mathit{getException}(\mathit{jcre}, \mathit{cl}) = \mathit{jcre}.\mathit{eInst}(\mathit{cl})\\
&H \models \mathit{eInst} \Leftrightarrow \forall \mathit{cl} \in \mathsf{RTException} : (\mathit{cl} \in \mathit{dom}(\mathit{eInst})) \wedge\\
&\qquad\qquad (\mathit{eInst}(\mathit{cl}) \in \mathit{dom}(H)) \wedge (o.\mathit{class} = \mathit{cl})\\
&\qquad\qquad (o.\mathit{owner} = \mathit{JCRE}) \wedge (o.\mathit{entryPoint} = \mathtt{tEP})\\
&\qquad\quad \text{where } o = H(\mathit{eInst}(\mathit{cl}))
\end{aligned}$$

5 The JCVM Firewall

The Java Card platform partitions the object system into separate protected object spaces called contexts. An applet's (owning) context is determined by the AID of the package where the applet was defined and the owning context of an object is determined by the owning context of the applet that created it. The JCVM firewall provides a security mechanism that prevents an object from being accessed by code running in a different context i.e. by an object with a different owning context. Object and field access among different instances of the same applet or instances of applets declared in the same package (group context) is allowed since they share the same context.

Prior to performing an access operation on an object, the JCVM performs a runtime check that depends on the instruction, the currently active context (*ctxt*), and the type and owner of the referenced object (o) [5, Chapter 6]. Additionally, store operations check the new value (H, v) being stored (H, v). The runtime checks are defined in Figure 3. The JCVM will throw an instance of the class `SecurityException` when a firewall violation has been detected. In general, access to an object is disallowed if the current context - the context of the current frame - is different from the owning context of the object that is being accessed (eq. 5). In addition the JCRE mantains its own context. This context has special system privileges and can access any object (eq. 4). Thus, in the simplest case, access of object o from context *ctxt* is allowed if any of the following conditions are met:

$$\mathit{isJCRE}(\mathit{ctxt}) \equiv (\mathit{ctxt} = \mathit{jcre}) \tag{4}$$
$$\mathit{sameContext}(\mathit{ctxt}, o) \equiv (\mathit{ctxt} = o.\mathit{owner}.\mathit{context}) \tag{5}$$

This is the case of the *checkGetField* predicate: an instance field of an object (o) can only be accessed if the currently active context (*ctxt*) is either the JCRE or the owning applet of the object. These conditions alone would make the firewall system too restrictive. The JCRE provides several mechanisms

$$checkGetField(ctxt, o) \equiv isJCRE(ctxt) \vee sameContext(ctxt, o)$$

$$checkPutStatic(ctxt, H, v) \equiv isJCRE(ctxt) \vee canStore(H, v)$$

$$\begin{aligned} checkPutField(ctxt, o, H, v) \equiv\ & isJCRE(ctxt) \vee (sameContext(ctxt, o) \wedge \\ & canStore(H, v)) \end{aligned}$$

$$checkThrow = checkInvokeDefinite = objectAccess$$

$$\begin{aligned} checkArrayStore(ctxt, app, a, H, v) \equiv\ & objectAccess(ctxt, app, a) \wedge \\ & (\neg isJCRE(ctxt) \Rightarrow canStore(H, v)) \end{aligned}$$

$$checkArrayLoad = checkInvokeVirtual = objectAccess$$

$$checkInvokeIface(ctxt, app, o, iface) \equiv$$
$$objectAccess(ctxt, app, o) \vee \begin{pmatrix} transAccess(ctxt, app, o) \\ \wedge\ isShareable(o, iface) \end{pmatrix}$$

$$checkCast(ctxt, app, o, \tau) \equiv \begin{cases} objectAccess(ctxt, app, o) \vee (\tau \in \mathsf{Interface} \\ \wedge\ transAccess(ctxt, app, o) \wedge isShareable(o, \tau)) \end{cases}$$

Fig. 3. JCVM Firewall Checks

to allow object access across contexts: global arrays, JCRE entry points and shareable interfaces. JCRE entry point objects - temporary and permanenet - and global arrays are objects owned by the JCRE that can be accessed from any context. We say that an object o has global access if the conditions defined in equation 6 are true. Objects with global access have a special restriction. References to objects designated as temporary JCRE entry points and global arrays cannot be stored in class fields, instance fields or array components, as defined by the *canStore* predicate in (7.

$$\begin{aligned} globalAccess(o) \equiv\ & (o \in \mathsf{ClassInst} \wedge o.entryPoint \neq \mathtt{no}) \vee \\ & (o \in \mathsf{ArrayObject} \wedge o.global) \end{aligned} \tag{6}$$

$$\begin{aligned} canStore(H, v) \equiv\ & (v \in \mathsf{ReferenceValue}) \Rightarrow canStoreObject(H(v)) \\ canStoreObject(o) \equiv\ & (o \in \mathsf{ArrayObject} \wedge \neg o.global) \vee \\ & (o \in \mathsf{ClassInst} \wedge o.entryPoint \neq \mathtt{temp}) \end{aligned} \tag{7}$$

The *objectAccess* predicate (equation 8) spells out the general conditions under which access to object o is allowed. The predicate includes an additional check for `CLEAR_ON_DESELECT` transient objects (*transAccess*, equation 9). Transient objects of `CLEAR_ON_DESELECT` type can only be accessed when the currently active context (*ctxt*) is the context of the currently selected apple

(app)[3]. We have:

$$\begin{aligned} &\mathit{objectAccess}(ctxt, app, o) \equiv \mathit{isJCRE}(ctxt) \vee \\ &\quad \left((\mathit{transAccess}(ctxt, app, o) \wedge \left(\begin{array}{c} \mathit{sameContext}(ctxt, o) \\ \vee\ \mathit{globalAccess}(o)) \end{array} \right) \right) \end{aligned} \tag{8}$$

$$\begin{aligned} &\mathit{transAccess}(ctxt, app, o) \equiv (o \in \mathsf{ArrayObject}) \Rightarrow \\ &\quad [(o.\mathit{transient} = \mathtt{CLEAR\_ON\_DESELECT}) \Rightarrow (ctxt = app)] \end{aligned} \tag{9}$$

Currently, only arrays with primitive type components and arrays with references to `Object` can be transient objects. It may seem an abuse of notation to include *transAccess* as part of the firewall checks of instructions that do not deal with arrays but, by doing so, we provide a general criteria for most firewall checks. Later versions of the Java Card platform may allow transient class instances. In the worse case, the *transAccess* condition is trivially true. Still, we would like to point out that it is possible to invoke methods of the `Object` class on an array.

Another mechanism that allows object access among contexts is the use of shareable interfaces. Methods of shareable interfaces can be invoked from one context even if the object implementing them is owned by an applet in another context. An interface $(iface)$ method can be invoked on object o if:

$$\mathit{isShareable}(o, iface) \equiv \begin{cases} \mathtt{Shareable} \in \mathit{implements}(o.\mathit{class}) \wedge \\ \mathtt{Shareable} \in \mathit{superInterfaces}(iface) \end{cases}$$

that is, if the object's class implements a `Shareable` interface i.e. o is a Shareable Interface Object, and if the interface in question extends the `Shareable` interface. The same conditions are used when the `checkcast` or `instanceof` instructions are applied to an interface type.

[3] The currently selected applet can be obtained from the JCRE element by calling the *getAppletContext* function

6 The Operational Semantics of the JCVM Language

6.1 The Evaluation Relation

The operational semantics of the JCVML (Carmel) is defined as a relation $\Rightarrow$ between program configurations. We say that the judgement $C \Rightarrow C'$ is true if configuration C' is the result of executing the current instruction (section 3.4) in configuration C.

We give the evaluation rules of Carmel in section 6.3. The evaluation rules detail the set of conditions under which the assertion $C \Rightarrow C'$ is true. The first condition in all the rules determines the instruction that is being executed: the current instruction. We show the process of reading and executing an instruction by presenting the evaluation rule for `nop`:

$$\frac{I = m.\mathit{instructionAt}(pc) \ \wedge \ I = \mathtt{nop}}{\langle G, SF{::}\langle own, m, pc, V, S\rangle\rangle \ \Rightarrow \ \langle G, SF{::}\langle own, m, pc+1, V, S\rangle\rangle}$$

The current instruction is obtained with $m.\mathit{instructionAt}(pc)$, where m and pc are the method and program counter kept by the current frame. The `nop` does not modify the state of the program except for the program counter. The program counter is incremented to point to the next instruction ($pc+1$)where execution should resume. Since the process of reading the instruction is the same for all rules, from now on we will consider $I = m.\mathit{instructionAt}(pc)$ as the default. In the example shown above, the first condition should be $I = \mathtt{nop}$.

The semantics presented in this section correspond to the semantics of the offensive virtual machine defined in [3]. Therefore, we only show the conditions and changes to the program state that result when the current execution succeeds. Successful execution of the instructions is ensured by the static conditions checked by the bytecode verified. Runtime violations of the semantics of the language - that can not be detected by the verifier - are reported by the JCVM by throwing an exception. The process of throwing an exception is explained in the next section.

6.2 Throwing Exceptions

The JCVM performs a series of runtime checks that preserve the semantic constraints of the language. These runtime checks appear in the evaluation rules as predicates prefixed with the word *check*. We have already defined the predicates that enforce the firewall conditions in section 5. The violation of a firewall predicate will result in a `SecurityException`. In this section, we define the rest of the predicates that determine the conditions for throwing a runtime exception. These are:

$$\begin{array}{ll}
\mathit{checkNull}(\mathit{loc}) & \equiv (\mathit{loc} \neq \texttt{null}) \\
\mathit{checkArithmetic}(\mathit{op}, v_1, v_2) & \equiv \mathit{op} \notin \{\texttt{div}, \texttt{rem}\} \vee v_2 \neq 0 \\
\mathit{checkNegative}(n) & \equiv (n \geq 0) \\
\mathit{checkCanCast}(H, \mathit{loc}, \tau) & \equiv H \vdash \mathit{loc} \ :\preceq \ \tau \\
\mathit{checkArrayBounds}(a, i) & \equiv (0 \leq i < a.\mathit{length})
\end{array}$$

We bound the runtime checks defined above to the the null pointer, arithmetic, negative array, class cast and array index out of bounds exceptions[4], respectively. The violation of any of these runtime checks will trigger the JCVM to throw the runtime exception associated with them. For example, if the *checkNull*(*loc*) condition is not true in a particular rule then the null pointer exception is thrown by the JCVM. If more than one runtime check is violated, then the exception associated to the runtime check that appears first in the rule will be thrown.

Let's have a look at the rule for `getfield` of Figure 5. The corresponding rule that describes the process of throwing a runtime exception is shown below. Once the runtime violation is found, the JCVM throws the JVRE owned exception object by first calling the *getException* function. Execution is resumed from the frame returned by *catchException*.

$$\frac{\begin{array}{l}
I = \texttt{getfield}\ f \ \wedge\ S = S'{::}\mathit{loc} \\
e = \begin{cases} \texttt{NullPointerException} & \text{if } \neg\mathit{checkNull}(\mathit{loc}) \\ \texttt{SecurityException} & \text{if } \neg\mathit{checkGetField}(\mathit{own.context}, H(\mathit{loc})) \end{cases} \\
(\mathit{jcre}', H', \mathit{loc}') = \mathit{getException}(\mathit{jcre}, H, e) \\
SF' = \mathit{catchException}(\mathrm{SF}{::}\langle \mathit{own}, m, \mathit{pc}, V, S\rangle, H', \mathit{loc}')
\end{array}}{\langle (\mathit{jcre}, K, H), SF{::}\langle \mathit{own}, m, \mathit{pc}, V, S\rangle\rangle \ \Rightarrow\ \langle (\mathit{jcre}, K, H'), SF'\rangle}$$

[4] `NullPointer`, `ArithmeticException`, `NegativeArraySizeException`, `ClassCastException` and `ArrayIndexOutOfBoundsException`.

6.3 Evaluation Rules

6.3.1 The Core Language

The core language supports instructions for stack manipulation, local variable access, numerical operation and branching. Stack instructions deal explicitly with the operand stack. They do not carry type information - with the sole exception of `push` that uses the operand type to put the right kind of value on top of the stack - and operate ignoring the type of data on the stack. Instead, stack instructions operate at the word level and their operands denote the number of words to be manipulated. For example, the execution of the `pop` n instruction results on the the top n words being popped from the operand stack. A more elaborate rule is shown below:

$$\frac{\begin{array}{c} I = \texttt{dup}\ n\ d\ \wedge\ S = S_3 : S_2 : S_1 \\ n = \mathit{nbWords}(S_1) \wedge d = \mathit{nbWords}(S_2 : S_1)\ \wedge\ S' = S_3 : S_1 : S_2 : S_1 \end{array}}{\langle G, SF{::}\langle own, m, pc, V, S\rangle\rangle\ \Rightarrow\ \langle G, SF{::}\langle own, m, pc+1, V, S'\rangle\rangle}$$

Execution of the `dup` instruction results on a new configuration where the top n words on the operand stack have been duplicated and inserted d words down from the top. When d equals 0, the top n words are copied and placed on top of the stack. The rules respect the restriction that preserve the integrity of two-word values by making sure that only whole values are moved. This is also true for the instructions that manipulate the array of local variables V where, as we pointed out in section 3.3, the special symbol $\perp$ is used to "block" the index that corresponds to the second word of an integer. The `load` instruction places the value of local variable i on top of the stack and `store` updates local variable i with the value found on top of the stack. The success of the rules is ensured by the bytecode verification process which we assume have taken place.

The `numop` instruction uses the top two elements of the stack - after removing them - as arguments to the numeric operation *op* and places the result back on the stack. The `goto` instruction jumps to the instruction found in address *addr*. The `if` instruction - shown below - jumps to *addr* only if the result of performing the comparison operation *cmp* between the top two elements of

$$\frac{I = \texttt{push}\ \ t\ \ c}{\langle G, SF{::}\langle own, m, pc, V, S\rangle\rangle \ \Rightarrow\ \langle G, SF{::}\langle own, m, pc+1, V, S{::}c^t\rangle\rangle}$$

$$\frac{I = \texttt{pop}\ \ n\ \wedge\ n = nbWords(S')}{\langle G, SF{::}\langle own, m, pc, V, S : S'\rangle\rangle \ \Rightarrow\ \langle G, SF{::}\langle own, m, pc+1, V, S\rangle\rangle}$$

$$\frac{\begin{array}{c} I = \texttt{swap}\ \ n_1\ \ n_2\ \wedge\ S = S_3 : S_2 : S_1 \\ n_1 = nbWords(S_1) \wedge n_2 = nbWords(S_2)\ \wedge\ S' = S_3 : S_1 : S_2 \end{array}}{\langle G, SF{::}\langle own, m, pc, V, S\rangle\rangle \ \Rightarrow\ \langle G, SF{::}\langle own, m, pc+1, V, S'\rangle\rangle}$$

$$\frac{\begin{array}{c} I = \texttt{numop}\ \ t\ \ op \\ checkArithmetic(op, v_1, v_2)\ \wedge\ r = applyBinary(op, v_1, v_2, t) \end{array}}{\langle G, SF{::}\langle own, m, pc, V, S{::}v_1{::}v_2\rangle\rangle \ \Rightarrow\ \langle G, SF{::}\langle own, m, pc+1, V, S{::}r\rangle\rangle}$$

$$\frac{I = \texttt{load}\ \ t\ \ i}{\langle G, SF{::}\langle own, m, pc, V, S\rangle\rangle \ \Rightarrow\ \langle G, SF{::}\langle own, m, pc+1, V, S{::}V[i]\rangle\rangle}$$

$$\frac{I = \texttt{store}\ \ t\ \ i}{\langle G, SF{::}\langle own, m, pc, V, S{::}v\rangle\rangle \ \Rightarrow\ \langle G, SF{::}\langle own, m, pc+1, V[i\ \mapsto\ v], S\rangle\rangle}$$

$$\frac{I = \texttt{inc}\ \ t\ \ i\ \ c\ \wedge\ r = V[i] + c}{\langle G, SF{::}\langle own, m, pc, V, S\rangle\rangle \ \Rightarrow\ \langle G, SF{::}\langle own, m, pc+1, V[i\ \mapsto\ r], S\rangle\rangle}$$

$$\frac{I = \texttt{goto}\ \ addr}{\langle G, SF{::}\langle own, m, pc, V, S\rangle\rangle \ \Rightarrow\ \langle G, SF{::}\langle own, m, addr, V, S\rangle\rangle}$$

Fig. 4. The Core Language Semantics

the stack is true, otherwise it continues to the next instruction.

$$\frac{\begin{array}{l} I = \texttt{if}\ \ t\ \ cmp\ \ \texttt{goto}\ \ addr \\ pc' = \begin{cases} addr & \text{if } applyComparison(cmp, v_1, v_2) \\ pc + 1 & \text{otherwise} \end{cases} \end{array}}{\langle\!\langle G, SF{::}\langle\!\langle own, m, pc, V, S{::}v_1{::}v_2\rangle\!\rangle\rangle\!\rangle \ \Rightarrow\ \langle\!\langle G, SF{::}\langle\!\langle own, m, pc', V, S\rangle\!\rangle\rangle\!\rangle}$$

The rest of evaluation rules for the instructions of the core language are shown in figure 4. We have not included the evaluation rules of the `if` and `op` instructions that handle null comparisons and unary operations. We have also excluded the `keyswitch` and `indexswitch` instructions. The complete formalisation of these instructions can be found in [9].

6.4 The Object Language

The instructions described in this section deal with the creation and manipulation of class instances. The evaluation rules for the instructions of the object language are shown in figure 5. The `new` *cl* instruction creates a new instance object of class *cl* by calling the *newObecjt* function defined in (1). The owner of the current frame is designated as the owner of the new object. The `getstatic` and `putstatic` instruction use the field id *f.id* to access the static field *f* in K. The rest of the instruction take the location of the object to be read, modified or inspected is taken from the stack. For example, in the case of the `putfield` instruction shown below,

$$\frac{\begin{array}{l} I = \texttt{putfield}\ \ f\ \wedge\ v' = fromStack(v, f.type) \\ checkNull(loc) \wedge checkPutField(own.context, o, H, v) \\ H' = updateElement(H, (loc, f.id), v') \end{array}}{\langle\!\langle G, SF{::}\langle\!\langle own, m, pc, V, S'{::}loc{::}v\rangle\!\rangle\rangle\!\rangle \ \Rightarrow\ \langle\!\langle G', SF{::}\langle\!\langle own, m, pc+1, V, S'\rangle\!\rangle\rangle\!\rangle}$$

the location of the object to be modified and the new value of field f are taken from the operand stack. A type conversion is performed, if needed, by the *fromStack* function (section 3.3). The new value is stored in field f of the resolved object o using the *updateElement* function (eq. 3). The `putfield this` and `getfield this` instructions (not included here) take the location of the object from $V[0]$, the location of the current object.

The semantics of `checkcast` and `instanceof` are given by the definition of $H \vdash loc\ :\preceq\ \tau_{\mathrm{r}}$ (Figure 2). The difference between `checkcast` and `instanceof`

Let $app = jcre.getAppletContext$

$$\frac{I = \texttt{new}\ \ cl \wedge cl \in \mathsf{Class} \qquad (H', loc) = newObject(own, cl, \texttt{no}, H)}{\langle\!\langle G, SF{::}\langle\!\langle own, m, pc, V, S\rangle\!\rangle\rangle\!\rangle \ \Rightarrow\ \langle\!\langle G', SF{::}\langle\!\langle own, m, pc+1, V, S{::}loc\rangle\!\rangle\rangle\!\rangle}$$

$$\frac{I = \texttt{getstatic}\ \ f \wedge v = toStack(K(f.id), f.type)}{\langle\!\langle G, SF{::}\langle\!\langle own, m, pc, V, S\rangle\!\rangle\rangle\!\rangle \ \Rightarrow\ \langle\!\langle G, SF{::}\langle\!\langle own, m, pc+1, V, S{::}v\rangle\!\rangle\rangle\!\rangle}$$

$$\frac{I = \texttt{putstatic}\ \ f \ \wedge\ checkPutStatic(own.context, H, v) \qquad v' = fromStack(v, f.type)\ \wedge\ K' = K[f.id\ \mapsto\ v']}{\langle\!\langle G, SF{::}\langle\!\langle own, m, pc, V, S{::}v\rangle\!\rangle\rangle\!\rangle \ \Rightarrow\ \langle\!\langle G', SF{::}\langle\!\langle own, m, pc+1, V, S\rangle\!\rangle\rangle\!\rangle}$$

$$\frac{I = \texttt{getfield}\ \ f \wedge checkNull(loc) \wedge o = H(loc) \qquad checkGetField(own.context, o) \wedge v = toStack(o.fieldValue(f.id), f.type)}{\langle\!\langle G, SF{::}\langle\!\langle own, m, pc, V, S'{::}loc\rangle\!\rangle\rangle\!\rangle \ \Rightarrow\ \langle\!\langle G, SF{::}\langle\!\langle own, m, pc+1, V, S'{::}v\rangle\!\rangle\rangle\!\rangle}$$

$$\frac{I = \texttt{checkcast}\ \ \tau_{\mathrm{r}}\ \wedge\ checkCanCast(H, loc, \tau_{\mathrm{r}}) \qquad loc \neq \texttt{null}\ \Rightarrow\ checkCast(own.context, app, \tau_{\mathrm{r}}, H(loc))}{\langle\!\langle G, SF{::}\langle\!\langle own, m, pc, V, S{::}loc\rangle\!\rangle\rangle\!\rangle \ \Rightarrow\ \langle\!\langle G, SF{::}\langle\!\langle own, m, pc+1, V, S{::}loc\rangle\!\rangle\rangle\!\rangle}$$

$$\frac{I = \texttt{instanceof}\ \ \tau_{\mathrm{r}} \wedge v = \begin{cases} 1\ loc \neq \texttt{null} \wedge H \vdash loc\ :\preceq\ \tau_{\mathrm{r}} \\ 0 \text{ otherwise} \end{cases} \qquad loc \neq \texttt{null}\ \Rightarrow\ checkCast(own.context, app, \tau_{\mathrm{r}}, H(loc))}{\langle\!\langle G, SF{::}\langle\!\langle own, m, pc, V, S{::}loc\rangle\!\rangle\rangle\!\rangle \ \Rightarrow\ \langle\!\langle G, SF{::}\langle\!\langle own, m, pc+1, V, S{::}v\rangle\!\rangle\rangle\!\rangle}$$

Fig. 5. The Object Language Semantics

is that the former throws a `ClassCastException` when the operation is not valid. Firewall security checks take as arguments the current context, the current applet context, the object being accessed and, in the case of `putstatic` and `putfield`, the value being stored.

The evaluation rules for the instructions of the method language are shown in figure 6. The `invokedefinite` instruction is used for static methods and for non-static methods that have been resolved statically. The `invokedefinite` instruction contains a resolved method as argument. No method search through the class hierarchy is needed, even if the original call was made to an inherited method. The method has been resolved statically i.e. before execution when loaded and linked.

The `invokevirtual` and `invokeinterface` instructions perform a method lookup based on the class of the object where the method is being invoked (target object). This search is performed the *methodIDLookup* function. All methods sharing the same signature are assigned the same method id. Given a method id and a class *cl*, the function *methodLookup*(id, cl) returns the first method structure with the same method id found in the superclass hierarchy of class *cl*. The search is defined by:

$$\begin{aligned}
&\mathit{methodLookup} : \mathsf{MethodID} \times \mathsf{Class}_\perp \rightarrow \mathsf{Method}_\perp \\
&\mathit{methodIDLookup}(id, \perp) = \perp \\
&\mathit{methodIDLookup}(id, cl) = m \Leftrightarrow m \in cl.\mathit{methods} \wedge m.id = id \\
&\mathit{methodIDLookup}(id, cl) = \mathit{methodIDLookup}(id, cl.\mathit{superclass}) \Leftrightarrow \\
&\qquad\qquad\qquad\qquad\qquad\qquad \forall m \in cl.\mathit{methods},\ m.id \neq id
\end{aligned}$$

An array may the target of an `invokevirtual` instruction. If that's the case then the search must be performed on the `Object` class structure. This check is performed by the *methodLookup* function which, given a method m and an object o, returns the correct method:

$$\mathit{methodLookup}(m, o) = \begin{cases} \mathit{methodIDLookup}(m'.id, \texttt{Object}) & o \in \mathsf{ArrayObject} \\ \mathit{methodIDLookup}(m'.id, o.\mathit{class}) & \text{otherwise} \end{cases}$$

We describe the execution of the `invokevirtual` instruction (see rule below). The top $n+1$ values - corresponding to the n method arguments and object location (`this`) - are popped from the operand stack of the current frame and placed in the local variables array of the new frame F'. The owner of the target object o is set as owner of the new frame. The new operand stack is set to empty and, if both runtime checks succeed, the new frame is pushed

$$\frac{\begin{array}{c} I = \texttt{invokedefinite}\ \ m' \ \wedge\ m'.\mathit{isStatic}\ \wedge\ n = |m'| \\ S_0 = S{::}v_1{::}\ldots{::}v_n \wedge V' = v_1{::}\ldots{::}v_n \wedge F' = \langle \mathit{own}, m', 0, V', \epsilon\rangle \end{array}}{\langle G, SF{::}\langle \mathit{own}, m, pc, V, S_0\rangle\rangle \ \Rightarrow\ \langle G, SF{::}\langle \mathit{own}, m, pc, V, S\rangle{::}F'\rangle}$$

$$\frac{\begin{array}{c} I = \texttt{invokedefinite}\ \ m' \ \wedge\ \neg m'.\mathit{isStatic} \\ n = |m'| \ \wedge\ S_0 = S{::}\mathit{loc}{::}v_1{::}\ldots{::}v_n \ \wedge\ \mathit{checkNull}(\mathit{loc}) \\ o = H(\mathit{loc}) \ \wedge\ \mathit{checkInvokeDefinite}(\mathit{own.context}, o) \\ V' = \mathit{loc}{::}v_1{::}\ldots{::}v_n \wedge F' = \langle \mathit{o.owner}, m', 0, V', \epsilon\rangle \end{array}}{\langle G, SF{::}\langle \mathit{own}, m, pc, V, S_0\rangle\rangle \ \Rightarrow\ \langle G, SF{::}\langle \mathit{own}, m, pc, V, S\rangle{::}F'\rangle}$$

$$\frac{\begin{array}{c} I = \texttt{invokeinterface}\ \ m' \ \wedge\ n = |m'| \ \wedge\ S_0 = S{::}\mathit{loc}{::}v_1{::}\ldots{::}v_n \\ \mathit{checkNull}(\mathit{loc}) \wedge o = H(\mathit{loc}) \ \wedge\ m_v = \mathit{methodLookup}(m', o) \\ \mathit{checkInvokeIface}(\mathit{own.context}, \mathit{jcre.getAppletContext}, m'.\mathit{classI}, o) \\ V' = \mathit{loc}{::}v_1{::}\ldots{::}v_n \wedge F' = \langle \mathit{o.owner}, m_v, 0, V', \epsilon\rangle \end{array}}{\langle G, SF{::}\langle \mathit{own}, m, pc, V, S_0\rangle\rangle \ \Rightarrow\ \langle G, SF{::}\langle \mathit{own}, m, pc, V, S\rangle{::}F'\rangle}$$

$$\frac{I = \texttt{return} \ \wedge\ F' = \langle \mathit{own}', m', pc', V', S'\rangle}{\langle G, SF{::}F'{::}\langle \mathit{own}, m, pc, V, S\rangle\rangle \ \Rightarrow\ \langle G, SF{::}\langle \mathit{own}', m', pc' + 1, V', S'\rangle\rangle}$$

$$\frac{I = \texttt{return}}{\langle G, \langle \mathit{own}, m, pc, V, S\rangle\rangle \ \Rightarrow\ \langle (G), \texttt{halt}\rangle}$$

Fig. 6. The Method Language Semantics

onto the call frame stack. The current method is now m_v and execution will continue at the first instruction of the method ($pc = 0$). A context switch occurs when a method of an object owned by a different context is invoked i.e. two consecutive frames have different contexts.

$$\frac{\begin{array}{c} I = \texttt{invokevirtual}\ \ m' \ \wedge\ n = |m'| \\ F = \langle own, m, pc, V, S{::}loc{::}v_1{::}\ldots{::}v_n\rangle \\ checkNull(loc)\ \wedge\ o = H(loc)\ \wedge\ m_v = methodLookup(m', o) \\ checkInvokeVirtual(own.context, jcre.getAppletContext, o) \\ V' = loc{::}v_1{::}\ldots{::}v_n\ \wedge\ F' = \langle o.owner, m_v, 0, V', \epsilon\rangle \end{array}}{\langle G, SF{::}F\rangle\ \Rightarrow\ \langle G, SF{::}\langle own, m, pc, V, S\rangle{::}F'\rangle}$$

Method return is handled by the `return` instruction. We have identified three cases. The first case,

$$\frac{\begin{array}{c} I = \texttt{return}\ \ t \\ F = \langle own, m, pc, V, S{::}v\rangle\ \wedge\ F' = \langle own', m', pc', V', S'\rangle \end{array}}{\langle G, SF{::}F'{::}F\rangle\ \Rightarrow\ \langle G, SF{::}\langle own', m', pc'+1, V', S'{::}v\rangle\rangle}\ ,$$

corresponds to the return from a method with return type different from `null`. The value v is popped from the operand stack of the current frame and pushed onto the operand stack of the frame of the invoker. The virtual machine then removes the current frame and returns control to the invoker. The other two cases - shown in figure 6 - correspond to `void` returns. The last rule handles handles a return from the last frame which, in turn, generates the special halt configuration.

6.6 The Array, Exception and Subroutine Language

The evaluation rules of the instructions that deal with the creation and manipulation of arrays, the explicit throwing of exceptions by the programmer and the call of (and return from) subroutines, are listed in Figure 7. A new array is created by the `new` τ instruction when the parameter τ denotes an array type. New arrays are initialised as non-global, non-transient objects by the *newArray* function (eq. 2). The owner of the current frame is set as the owner of the new array. The location of the new array is placed on top of the operand stack of the current frame. The other array instructions take their arguments - array location and index - from the operand stack, as shown by

Let $app = jcre.getAppletContext$:

$$\frac{I = \texttt{new}\ \ \tau\texttt{[ ]} \wedge checkNegative(n) \qquad (H', loc) = newArray(own, \tau, n, \texttt{false}, \texttt{NOT\_TRANSIENT}, H)}{\langle\!\langle G, SF{::}\langle\!\langle own, m, pc, V, S{::}n\rangle\!\rangle \ \Rightarrow\ \langle\!\langle G', SF{::}\langle\!\langle own, m, pc+1, V, S{::}loc\rangle\!\rangle}$$

$$\frac{I = \texttt{arraylength} \ \wedge\ checkNull(loc) \qquad checkArrayLoad(own.context, app, H(loc))\ \wedge\ v = H(loc).length}{\langle\!\langle G, SF{::}\langle\!\langle own, m, pc, V, S{::}loc\rangle\!\rangle \ \Rightarrow\ \langle\!\langle G, SF{::}\langle\!\langle own, m, pc+1, V, S{::}v\rangle\!\rangle}$$

$$\frac{\begin{array}{c} I = \texttt{arraystore}\ \ t\ \wedge\ checkNull(loc)\ \wedge\ a = H(loc) \\ checkArrayBounds(a, i)\ \wedge\ checkArrayStore(own.context, app, a, H, v) \\ v' = fromStack(v, t)\ \wedge\ H' = updateElement(H, (loc, i), v') \end{array}}{\langle\!\langle G, SF{::}\langle\!\langle own, m, pc, V, S{::}loc{::}i{::}v\rangle\!\rangle \ \Rightarrow\ \langle\!\langle G', SF{::}\langle\!\langle own, m, pc+1, V, S\rangle\!\rangle}$$

$$\frac{\begin{array}{c} I = \texttt{throw} \\ checkNull(loc) \wedge checkThrow(own.context, H(loc)) \\ SF' = catchException(SF{::}\langle\!\langle own, m, pc, V, S\rangle\!\rangle, H, loc) \end{array}}{\langle\!\langle G, SF{::}\langle\!\langle own, m, pc, V, S{::}loc\rangle\!\rangle \ \Rightarrow\ \langle\!\langle G, SF'\rangle\!\rangle}$$

$$\frac{I = \texttt{jsr}\ \ addr}{\langle\!\langle G, SF{::}\langle\!\langle own, m, pc, V, S\rangle\!\rangle \ \Rightarrow\ \langle\!\langle G, SF{::}\langle\!\langle own, m, addr, V, S{::}pc+1\rangle\!\rangle}$$

$$\frac{I = \texttt{ret}\ \ i\ \wedge\ pc' = V[i]}{\langle\!\langle G, SF{::}\langle\!\langle own, m, pc, V, S\rangle\!\rangle \ \Rightarrow\ \langle\!\langle G, SF{::}\langle\!\langle own, m, pc', V, S\rangle\!\rangle}$$

Fig. 7. The Array, Exception and Subroutine Language Semantics

the rule of `arrayload` below.

$$\frac{\begin{array}{l} I = \texttt{arrayload}\ t \ \wedge\ \mathit{checkNull}(loc) \ \wedge\ a = H(loc) \\ \mathit{checkArrayBounds}(a, i) \ \wedge\ v = \mathit{toStack}(a[i], t) \\ \mathit{checkArrayLoad}(own.context, jcre.getAppletContext, a) \end{array}}{\langle G, SF{::}\langle own, m, pc, V, S{::}loc{::}i\rangle\rangle \ \Rightarrow\ \langle G, SF{::}\langle own, m, pc+1, V, S{::}v\rangle\rangle}$$

It is worth pointing out that the *toStack* and *fromStack* functions are required to transfer data from the stack to an array, an vice-versa.

The rest of the rules deal with the Exception and Subroutine language. The `throw` instruction is used by the programmer to throw the exception object placed on top of the stack. The location *loc* on top of the stack must be a reference to an object that is an instance of class `Throwable` or of a subclass of `Throwable`. The process of catching the exception referenced by *loc* is performed by $\mathit{catchException}(SF{::}\langle own, m, pc, V, S\rangle, H, loc)$, where *loc* has been popped from the operand stack of the top frame. The semantics of *catchException* is described in section 4. The `jsr` instruction jumps to the subroutine address indicated by its operand and places the address of the next instruction on top of the stack. The `ret` instruction jumps to the address stored in local variable i.

7 Related Work

Since its official release, the Java language has attracted considerable interest from the research community. Hartel and Moreau [10] provide a comprehensive review of the substantial amount of work dedicated to the study of each of the main components of Java: the Java Language[11–13], the Java Virtual Machine (JVM)[14–16] and Java to JVM compiler [17,18]; and most of its interesting features: object-orientation, dynamic class loading, garbage collection, multi-threading, its type system and bytecode verification. However, as pointed out in [10], no single attempt has been made at specifying full Java, the full JVM, or the full compiler. The type system used in this paper is influenced by the work of Freund and Mitchell [19].

The smaller size and complexity of the Java Card platform simplifies the task of formalising the whole system or some of its parts. Most research of the semantics of Java Card has been produced by - or is related to - the formal methods and program verification community. Formal specifications of the semantics of Java Card can be traced back to [20] (virtual machine) and [21] (source code). Hartel et al.provide a complete specification of the

Java Secure Procesor (JSP) [20], a precursor of the JCVM. JSP is a virtual machine designed to fit on a smart card and, as such, does not support multi-threading, garbage collection and exception handling. The specifications are defined and validated using the LETOS tool [22]. Attali et al.adapt their work with Java and the Centuar tool[11] to Java Card and provide a programming environment for Java Card applications [21]. The research group at Gemplus has produced some work related to the application of the B-method [23] to the formalisation of the semantics of Java Card [24–26] .

CertiCartes[27–29] constitutes one of the most in-depth machine-checked accounts of the Java Card platform up to date. It contains formal executable specifications written in Coq of a defensive JCVM, an offensive JCVM and an abstract JCVM together with the specification of a Java Card Bytecode Verifier (BCV) presented as a data-flow analyser based on the abstract virtual machine

The LOOP project [30] project is involved in the application of formal methods to object oriented languages. The aim is to specify and verify properties of classes in object-oriented languages, using proof tools like PVS [31] and Isabelle/HOL [32].Its main contribution to the formalisation of the semantics of Java Card has been the formal specification of the Java Card API [33,34]. The complete specification of the Java Card API (framework and security classes) in JML and ESC/Java can be found in [35].

8 Conclusions and Future Work

The work presented in this paper has been developed in the context of the Secsafe project. The Secsafe project is concerned with the application of static analysis technology to the validation of security and safety aspects of realistic languages and applications. The project has focused a substantial part of its efforts on the development of methods that can be applied to the domain of smart cards and, in particular, Java Card and its applications (applets). In this context, a formalisation of the semantics of Java Card provides the framework where we can specify security properties, derive control and data flow analyses that safely approximate such properties, and prove the analyses correct.

Our first step have been to formalise the JCVM by defining the semantics of Carmel, a complete representation of the JCVML. The results of this work are presented in this paper. A more in-depth account of this specification can be found at our technical report [9].

We have seen that the specification presented in this paper relies on elements of the JCRE. Other important features of the JC platform are implemented by the API. In order for this specification to be useful, we need to placed it in context with the runtime environment and formalise its interaction with the off-card applications. The next step has been to provide a formal specification of the JCRE and the API. This is work in progress and can be found in [8].

Most work on formalising the semantics of Java card - as seen in the previous section - has relied on automated tools for mathematical specification and program verification. We intend to translate our specification to the PVS [31] system and use it as a base to prove the correctness of some of the analyses developed in the project.

References

[1] R. Marlet, Syntax of the JCVM language to be studied in the SecSafe project, Tech. Rep. SECSAFE-TL-005, v1.7, Trusted Logic, secSafe public (May 2001).

[2] Z. Chen, Java Card technology for Smart Cards: architecture and programmer's guide, Java series, Addison-Wesley, Reading, MA, USA, 2000.

[3] Sun Microsystems, Inc., Palo Alto, CA 94303 USA, The Java Card^{TM} 2.1.1 Virtual Machine Specification, revision 1.0 Edition (May 18 2000).

[4] Sun Microsystems, Inc., Palo Alto/CA, USA, Java Card 2.1.1 Platform API Specification (May 2000).
URL `java.sun.com/products/javacard/javacard21.html`

[5] Sun Microsystems, Inc., Palo Alto, CA 94303 USA, The Java Card^{TM} 2.1.1 Runtime Environemnt (JCRE) Specification, revision 1.0 Edition (May 18 200).

[6] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, 2nd Edition, Addison-Wesley Publishing Co., Reading, MA, 2000.

[7] R. R. Hansen, Flow logic for JVML, Tech. Rep. SECSAFE-DAIMI-005, v2.1, DAIMI, secSafe Internal (Feb. 2001).

[8] I. Siveroni, Formal specification of Java Card Runtime Environment and API, Tech. Rep. SECSAFE-ICSTM-006, Imperial College London, http://www.doc.ic.ac.uk/˜siveroni/secsafe (2002).

[9] I. Siveroni, C. Hankin, A proposal of the JCVMLe operational semantics, Tech. Rep. SECSAFE-ICSTM-001, Imperial College London, http://www.doc.ic.ac.uk/˜siveroni/secsafe (2002).

[10] P. H. Hartel, L. A. V. Moreau, Formalizing the safety of Java, the Java Virtual Machine and Java Card, ACM Computing Surveys 33 (4) (2001) 517–558.

[11] I. Attali, D. Caromel, M. Russo, A formal executable semantics for java, in: Proceedings of Formal Underpinnings of Java - An OOPSLA'98 Workshop, Vancouver, Canada, 1998.

[12] J. Alves-Foss, L. F., Dynamic denotational semantics of Java, in: J. Alves-Foss (Ed.), Formal Syntax and Semantics of Java, Vol. 1523 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 201–240.

[13] J. van den Berg, B. Jacobs, The LOOP compiler for Java and JML, in: T. Margaria, W. Yi (Eds.), Tools and algorithms for the construction and analysis of systems: 7th international conference, TACAS 2001, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Vol. 2031 of Lecture Notes in Computer Science, 2001, pp. 299–??

[14] R. M. Cohen, The defensive Java Virtual Machine specification, version 0.5, Technical
report, Computational Logic Inc., http://www.cli.com/software/djvm/ (May 1997).

[15] P. Bertelsen, Semantics of Java byte code, Technical report, Department of Information Technology, Technical University of Denmark, http://www.dina.kvl.dk/~pmb/ (Mar. 1997).

[16] K. Stephenson, Towards an algebraic specification of the Java Virtual Machine, in: M. B., T. J. V. (Eds.), Prospects for Hardware Foundations. ESPRIT Working Group 8533. NADA - New Hardware Design Methods. Survey Chapters, Vol. 1546 of Lecture Notes in Computer Science (LNCS), Springer-Verlag, 1998, pp. 236–277.

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

[18] M. Strecker, Formal verification of a Java compiler in Isabelle, in: Proc. Conference on Automated Deduction (CADE), Vol. 2392 of Lecture Notes in Computer Science, Springer Verlag, 2002, pp. 63–77.

[19] S. N. Freund, J. C. Mitchell, A formal framework for the Java Bytecode Language and Verifier, in: Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA '99, Vol. 34(10) of SIGPLAN Notices, ACM Press, Denver, Colorado, USA, 1999, pp. 147–166.

[20] P. H. Hartel, M. J. Butler, M. Levy, The operational semantics of a Java secure processor, in: J. Alves-Foss (Ed.), Formal Syntax and Semantics of Java, Vol. 1523 of Lecture Notes in Computer Science, Springer-Verlag, Berlin Germany, 1999, pp. 313–352.

[21] I. Attali, D. Caromel, C. Courbis, L. Henrio, H. Nilsson, Smarttools for Java Card, in: J. Domingo-Ferrer, D. Chan and A. Watson (Eds): 4th Smart Card Research and Advanced Application Conf. (CARDIS), Kluwer Academic Publishers, 2000, pp. 155–174.

[22] P. H. Hartel, LETOS - a lightweight execution tool for operational semantics, Software - Practice and Experience 29 (15) (1999) 1379–1416.

[23] J.-R. Abrial, The B-Book: Assigning Programs to Meanings, Cambridge University Press, 1996.

[24] J.-L. Lanet, A. Requet, Formal proof of smart card applets correctness, in: J.-J. Quisquater, B. Schneier (Eds.), Smart card research and applications: third international conference, CARDIS'98, Louvain-la-Neuve, Belgium, September 1998: proceedings, Louvain-la-Neuve, (be), 1998, p. ??

[25] A. Requet, A B model for ensuring soundness of the Java card virtual machine, in: S. Gnesi, I. Schieferdecker, A. Rennoch (Eds.), 5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems, FMICS 2000, GMD Report No. 91, Berlin, 2000.

[26] L. Casset, Development of an embedded verifier for Java card byte code using formal methods, in: L.-H. Eriksson, P. A. Lindsay (Eds.), FME 2002: formal methods-getting it right: International Symposium of Formal Methods Europe, Vol. 2391 of Lecture Notes in Computer Science, 2002, pp. 290–??

[27] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, S. M. de Sousa, A formal executable semantics of the JavaCard platform, in: D. Sands (Ed.), Programming languages and systems: 10th European Symposium on Programming, ESOP 2001, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Vol. 2028 of Lecture Notes in Computer Science, 2001, pp. 302–??

[28] G. Barthe, G. Dufay, L. Jakubiec, S. M. de Sousa, A formal correspondence between offensive and defensive JavaCard virtual machines, in: A. Cortesi (Ed.), Verification, model checking and abstract interpretation: third international workshop, VMCAI 2002, Vol. 2294 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 32–??

[29] G. Barthe, P. Courtieu, G. Dufay, S. M. de Sousa, Tool-assisted specification and verification of the JavaCard platform, in: H. Kirchner, C. Ringeissen (Eds.), Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Vol. 2422 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 41–??

[30] The LOOP project, http://www.cs.kun.nl/ bart/LOOP/.

[31] The PVS specification and verification system, http://pvs.csl.sri.com/.

[32] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Vol. 2283 of LNCS, Springer-Verlag, 2002.

[33] E. Poll, J. van den Berg, B. Jacobs, Formal specification of the JavaCard API in JML, in: J. Domingo-Ferrer, D. Chan and A. Watson (Eds): 4th Smart Card Research and Advanced Application Conf. (CARDIS), Kluwer Acad. Publ., 2000, pp. 135–154.

[34] H. Meijer, E. Poll, Towards a full formal specification of the JavaCard API, in: I. Attali, T. Jensen (Eds.), Smart card programming and security: International Conference on Research in Smart Cards, E-smart 2001, Vol. 2140 of Lecture Notes in Computer Science, 2001, pp. 165–??

[35] E. Hubbers, E. Poll, ESC/Java and JML specifications of the Java Card API 2.1.1, http://www.cs.kun.nl/~erikpoll/publications/jc211_specs.html.