Formalisation of the Semantics of Java Card

Author:	Igor Siveroni
Date:	October, 2003
Number:	SECSAFE-ICSTM-014
Version:	2.0
Classification:	Public
Status:	Stable
Comment:	Replaces SECSAFE-ICSTM-006

1 Introduction

We present a formal specification of the operational semantics of the Java Card JCRE and API. The Java Card platform is defined by three specifications: The *Java Card Virtual Machine* (JCVM) *Specification* [6], the *Java Card Application Programming Interface* (API) [4], and the *Java Card Runtime Environment* (JCRE) *Specification* [5]. Even though the JCVM and API are presented as separate entities (and specifications) they are not independent. On the contrary, they are related in many ways and several actions of the API concern features of the JCVM, and vice-versa. Most research on formalisations of the JCVM and the Java Card API have appeared as separate formal specifications. Our work attempts to close this gap by presenting an specification of the operational semantics of Java Card that includes the JCRE and its components: the JCVM and the Java Card API.

This work 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.

One of the goals of the project is to provide static analyses that verify security and safety aspects of Java Card applets. Examples show that Java Card applets make heavy use of the API. The question of what code is going to be analysed, and how much of it is available, becomes crucial. Even if we consider the case where the code of the API methods is available for analysis we have to deal with the fact that some of these methods are native. Furthermore, the analysis of the non-native methods would represent a source of imprecision, deeming most of the analyses results useless. The combination of these issues gives us another reason to provide a complete specification of the Java card API. By assuming that the code of all API methods respect this formal specification, we can replace the analysis of an API method code with an abstraction derived from the same specification improving, in this way, the precision of the analysis of the whole applet.

This document comprises the specification of the semantics of Java Card as a small-step transition system that reacts to events sent by the host, and the specification of the `java.lang` and `javacard.framework` packages. We cover most of the important JCRE and API features: exceptions (section 6), applets and AIDs (section 7), transactions (section 8) and transient objects (section 9). We apply the framework used by the semantics to the specification of the most relevant API methods.

2 Java Card State

Java Card is a reactive system that is made of two components: card and host. Activity in the card is affected by the occurrence of events generated by the host. An event is generated and sent to the card every time the card is placed into the card acceptance device (**powerOn** event), the host sends a command (**Cmd**(*cmd*) event), a card tear-off occurs (**powerOff** event) or the host resets the card (**reset** event). The complete list of events is:

$$ev \in \mathsf{Event} ::= \textbf{reset} \mid \textbf{powerOff} \mid \textbf{powerOn} \mid \textbf{Cmd}(cmd) \mid \textbf{noEvent}$$

where *cmd* is an APDU command as defined below in section 2.1.

The semantics of the Java Card is defined as a transition 3 system between card configurations. The judgement $card \longrightarrow^{ev} card'$ reasons about the execution of one evaluation step on current configuration *card* under the occurrence of event ev. The occurrence of **noEvent** corresponds to the normal execution of one evaluation step when no interrupting event has occurred or when the card is waiting for a message from the host.

The Java Card configuration $\langle s, cad, J, vm, H\rangle$ is made of five components: the card state, the Card Acceptance Device (CAD), the JCRE, the Java Card virtual machine (JVCM) and the heap.

$$\begin{array}{rcl}
card \in \mathsf{Card} & = & \mathsf{CardState} \times \mathsf{CAD} \times \mathsf{JCRE} \times \mathsf{JCVM} \times \mathsf{Heap} \\
card & ::= & \langle s, cad, J, vm, H\rangle \\
\mathsf{CardState} & = & \mathtt{Off} \mid \mathtt{Reset}(r) \mid \mathtt{Ready} \mid \mathtt{Reply} \mid \mathsf{CmdState} \\
\mathsf{CmdState} & ::= & \mathtt{Select}(l) \mid \mathtt{Process}(p) \\
p \in \mathsf{ProcessState} & ::= & \{\mathtt{start}, \mathtt{process}\} \\
l \in \mathsf{SelectState} & ::= & \{\mathtt{start}, \mathtt{deselect}, \mathtt{select}\} \\
r \in \mathsf{ResetState} & ::= & \{\mathtt{start}, \mathtt{abort}\}
\end{array}$$

2.1 APDU Messages and the CAD

Communication between card and host is performed using messages which conform to the Application Protocol Data Unit (APDU) format, as defined in ISO specification 7816-4. There are two kinds of APDU messages: command and response. The host triggers the communication process by sending an APDU command to the card. Commands are received and inspected by the JCRE. Depending on its type, a command may be handled entirely by the JCRE or may involve the execution of code of one of the downloaded CAP files. Once the command is processed, the JCRE sends a response APDU message to the host.

The CAD is modelled by the CAD domain. The elements of the domain used by the top-level Java Card semantics are:

$$\begin{array}{rcl}
cad & \in & \mathsf{CAD} \\
\mathsf{MsgAPDU} & = & \mathsf{CommandAPDU} \cup \mathsf{RespondeAPDU} \cup \{\mathtt{NoMsg}\} \\
message & \in & \mathsf{CAD} \rightarrow \mathsf{MsgAPDU} \\
copyCmd & \in & \mathsf{CAD} \rightarrow (\mathsf{CommandAPDU} \rightarrow \mathsf{CAD}) \\
sendHost & \in & \mathsf{CAD} \rightarrow \mathsf{CAD}
\end{array}$$

where *message* stores the APDU message currently being processed, the *copyCmd* function copies a command from the host to the CAD, and the *sendHost* function sends the response stored in the CAD to the host. Other components of the CAD domain are explained in section 10.

2.2 The JCRE element

The JCRE manages the resources of the card much like an operating system. The JCRE element models the main aspects of the JCRE. It is specified as a domain made of several components that individually tackle a specific (or set of related) feature(s) of the Java card System. The JCRE components are:

$$\begin{array}{rcl} J & \in & \mathsf{JCRE} \\ \mathit{exceptionCmp} & \in & \mathsf{JCRE} \to \mathsf{ExceptionCmp} \\ \mathit{appletCmp} & \in & \mathsf{JCRE} \to \mathsf{AppletCmp} \\ \mathit{transactionCmp} & \in & \mathsf{JCRE} \to \mathsf{TransactionCmp} \\ \mathit{apduCmp} & \in & \mathsf{JCRE} \to \mathsf{APDUCmp} \end{array}$$

A complete specification of each component is given in the remaining of this document. Section 6 is dedicated to the Exceptions component, section 7 describes AppletCmp, section 8 does the same with TransactionCmp and, finally, section 10 provides a detailed description of APDUCmp.

2.3 The JCVM and the Heap

The JCVM [6] is the part of the Java Card that is in charge of the execution of Java Card programs i.e. applets. We model the JCVM as an element of the JCVM domain. The state of the JCVM is determined by the values of two fields: *program* and *config*, where *program* contains the set of packages loaded into the card and *config* describes the state of execution of the current applet. A third element, the *eval* function, models the operational semantics of the machine. The complete specification of the operationals semantics of the JCVM, and details on some of the domains used in this document, can be found in [3, 2].

The JCVM domain is defined as follows:

$$\begin{array}{rcl} \mathsf{JCVM} & = & (\mathit{program} : \mathsf{Program}) \times (\mathit{config} : \mathit{Config}) \times \\ & & (\mathit{eval} : (\mathsf{CAD} \times \mathsf{JCRE} \times \mathsf{Heap}) \to (\mathsf{JCVM} \times \mathsf{CAD} \times \mathsf{JCRE} \times \mathsf{Heap})) \end{array}$$

There are three types of JCVM configurations: running (*RConfig*), exception (*EConfig*) and return (*HConfig*) configuration. A running configuration indicates that execution of a method - and applet - is still in progress while the other two indicate termitation, either due the presence of an uncaught exception or due to succesful method return. JCVM configurations are defined as follows:

$$\begin{array}{rcl} C \in \mathit{Config} & = & \mathit{RConfig} + \mathit{EConfig} + \mathit{HConfig} \\ \mathit{RConfig} & ::= & \mathtt{Run}(F{::}SF) \text{ where } F \in \mathsf{Frame} \wedge SF \in \mathsf{Frame}^* \\ \mathit{EConfig} & ::= & \mathtt{Exc}(loc) \text{ where } loc \in \mathsf{Location} \\ \mathit{HConfig} & ::= & \mathtt{Ret}(v) \text{ where } v \in \mathsf{Value} \end{array}$$

A Running configuration $\mathtt{Run}(F{::}SF)$ describes the state of execution of the current applet by keeping track of the call stack $F{::}SF$. A frame describes describes the state of execution of a given method. The call stack keeps track of the sequence of method calls SF that lead to the execution of the current method F. The call stack is made of at least one frame, the current frame F. The frames in SF are said to be in suspended execution, awaiting for the return of the method they invoked. A frame is defined by:

$$\begin{array}{rcl} F \in \mathsf{Frame} & = & \mathsf{Owner} \times \mathsf{Method} \times \mathsf{PCounter} \times \mathsf{LocalVar} \times \mathsf{OperandStack} \\ \mathsf{Frame} & ::= & \langle own, m, pc, V, S \rangle \end{array}$$

where $\langle own, m, pc, V, S \rangle$ denotes the frame owned by $own \in \mathsf{Owner}$ that is about to execute the instruction pointed by pc in method m with array of local variables $V \in [\mathsf{Value}]$ and operand stack $S \in \mathsf{Value}^*$. Execution of the JCVM starts by first setting up the call stack to an initial configuration and then calling *eval*. An initial configuration is a running configuration (*RConfig*) such that:

$$\begin{array}{ll} vm.\mathit{config} = \mathtt{Run}(\langle o.own, m, 0, V, \epsilon \rangle) & \\ vm.\mathit{eval}(cad, J, H) = \langle vm', cad', J', H' \rangle & \quad \text{Evaluation - one instruction at a time} \end{array}$$

eval executes the current instruction of the JCVM, that is, the instruction addressed by the frame on top of the call stack. It takes as arguments and returns new versions of the CAD, the JCRE and the heap. The *eval*

$$\frac{(ev = \mathbf{powerOn}) \wedge \\ (card' = \langle \mathtt{Reset(start)}, cad, J, vm, H\rangle)}{\langle \mathtt{Off}, cad, J, vm, H\rangle \longrightarrow^{ev} card'} \qquad \frac{(ev = \mathbf{powerOff}) \wedge (s \neq \mathtt{Off})}{\langle s, cad, J, vm, H\rangle \longrightarrow^{ev} \langle \mathtt{Off}, cad, J, vm, H\rangle}$$

$$\frac{(ev = \mathbf{NoEvent}) \wedge \\ (card.state = \mathtt{Ready})}{card \longrightarrow^{ev} card} \qquad \frac{(ev = \mathbf{Cmd}(cmd)) \wedge (s' = commandState(cmd)) \\ (cad' = cad.copyCmd(cmd)) \wedge \\ (card' = \langle s', cad', J, vm, H\rangle)}{\langle \mathtt{Ready}, cad, J, vm, H\rangle \longrightarrow^{ev} card'}$$

$$\frac{(ev = \mathbf{NoEvent}) \wedge \\ (card.state \in \mathsf{CmdState}) \wedge \\ (card \xrightarrow{cmd} card')}{card \longrightarrow^{ev} card'} \qquad \frac{(ev = \mathbf{NoEvent}) \wedge \\ (cad' = cad.sendHost) \wedge \\ (card' = \langle \mathtt{Ready}, cad', J, vm, H\rangle)}{\langle \mathtt{Reply}, cad, J, vm, H\rangle \longrightarrow^{ev} card'}$$

$$\frac{(ev = \mathbf{reset}) \wedge \\ (card' = \langle \mathtt{Reset(start)}, cad, J, vm, H\rangle)}{\langle s, cad, J, vm, H\rangle \longrightarrow^{ev} card'} \qquad \frac{(ev = \mathbf{NoEvent}) \wedge \\ (\langle \mathtt{Reset}(r), cad, J, vm, H\rangle \xrightarrow{rst} card')}{\langle \mathtt{Reset}(r), cad, J, vm, H\rangle \longrightarrow^{ev} card'}$$

Figure 1: Java Card Semantics

function, when applied to a running configuration, leaves the call stack in three possible states: a running configuration, an exception configuration or a return configuration. An exception configuration $\mathtt{Exc}(()loc)$ contains the location *loc* of the uncaught exception object thrown during the execution of the program. The return configuration $\mathtt{Ret}(v)$ contains the value v returned by the mthod of the initial frame. We write $\mathtt{Ret}()$ if the return type of the entry method is `void`.

The heap H is a mapping from memory locations to class instances or array objects. A complete description can be found in [3, 2].

3 Java Card Semantics: Transition System

The semantics of Java Card is defined as a transition system between card configurations. The judgement $card \longrightarrow^{ev} card'$ reasons about the execution of configuration *card* under the occurrence of event *ev*. The state of a configuration may involve many atomic operations. Thus, $card \longrightarrow^{ev} card'$ is not a function since the occurrence of certain events, such as **powerOff**, can occur at any point during the execution of a state (between atomic operations), resulting in more then one possible state. In that sense, the semantics presented in this document can be considered as a big-step semantics.

The rules that define $card \longrightarrow^{ev} card'$ are listed in Figure 1.

3.1 Power Loss and Reset

Power loss occurs when the card is withdrawn from the CAD or if there is some other mechanical or electrical failure. Power loss is described in the semantics by the occurrence of the **powerOff** event. It can occur at any time and the resulting state is always `Off`.

When power is re-applied to the card the JCRE performs a series of reset operations. Card reset can also be triggered by the host via a **reset** event, as shown by the respective rule in Figure 1. Card reseting is represented by the `Reset` state and is reached after the occurrence of the **powerOn** event, if the previous card state is `Off`, or after the **reset** event.

Card reset involves the execution of at most two atomic steps:

- First, transient data is reset to its default value. If there was a transaction in progress when power was lost (or reset occurred), the next step of the reset operation will abort the transaction. If no transaction was in progress, the applet that was selected (if any) when power was lost (or reset occurred) becomes implicitly deselected by setting the currently selected applet to $\perp$, and the card state is set to `Ready`.

$$\begin{array}{l}\langle \texttt{Reset(start)}, cad, J, vm, H\rangle \xrightarrow{rst} card' \Leftrightarrow \\ \quad H' = clearTransient(H, \texttt{CLEAR\_ON\_RESET}) \wedge T = J.transactionCmp \\ \quad card' = \begin{cases} \langle \texttt{Reset(abort)}, cad, J, vm, H'\rangle & \text{if } T.transactionDepth = 1 \\ \langle \texttt{Ready}, cad, J[appletCmp \mapsto A'], vm, H'\rangle & \text{otherwise} \end{cases} \\ \qquad \text{where } A = J.appletCmp \wedge A' = A[current = \perp] \end{array} \tag{1}$$

- The abort transaction operation is performed, the selected applet (if any) is deselected and the card state is set to `Ready`.

$$\begin{array}{l}\langle \texttt{Reset(abort)}, cad, J, vm, H\rangle \xrightarrow{rst} card' \Leftrightarrow \\ \quad A = J.appletCmp \wedge T = J.transactionCmp \wedge T.transactionDepth = 1 \wedge \\ \quad \langle T', H', K'\rangle = T.abort(H'', K) \ \wedge A' = A[current = \perp] \\ \quad card' = \langle \texttt{Ready}, cad, J[transactionCmp \mapsto T'][appletCmp \mapsto A'], vm, H'\rangle \end{array} \tag{2}$$

The resulting state, the `Ready` state, should have no applet selected. All transient data should have been reset and the interrupted transaction, if any, aborted. The reset operation can be interrupted by power loss or reset.

We have not modelled the case when the JCRE implements default applet selection. With default applet selection, upon reset, the JCRE selects the designated default applet and invokes the default applet's `select` method. Its inclusion would involve the addition of extra steps to make room for the execution of the JCVM. It's exclusion makes the specification more succinct while maintaining the JCRE main features.

3.2 APDU Command Processing

$$card.processAPDU(ev) = \begin{cases} card.processSelect(ev) & \text{if } card.cad.msg.isSelect \\ \ldots & \\ card.processCommand(ev) & \text{otherwise} \end{cases} \tag{3}$$

3.2.1 Select APDU

An applet remains in a suspended state until is explicitly selected. Only after it has been selected is an applet ready to receive and process APDU commands. Applet selection is triggered by the host by sending a SELECT APDU command with the AID bytes of the applet to be selected. The format of the SELECT APDU command is:

CLA	INS	P1	P2	Lc	Data Field
0x0	0xA4	0x4	0x0	Length AID	AID bytes

We define the *isSelect* function that determines if a message is a well-formed Select APDU command:

$$\begin{array}{ll} isSelect : \mathsf{CommandAPDU} \rightarrow \mathsf{boolean} & \\ msg.isSelect \Leftrightarrow & (msg.CLA = 0\text{x}0) \wedge (msg.INS = 0\text{xA}4) \\ & (msg.P1 = 0\text{x}4) \wedge (msg.P2 = 0\text{x}0) \end{array}$$

The actions performed by the JCRE are described in equation ??. First, the JCRE checks if the AID bytes encoded in the data part of the APDU command correspond to any of the applets in the card. If there is a match then the JCRE deselects the current applet, if any, by invoking the `Applet.deselect` method on the applet. Once the current applet has been succesfully deselected, the JCRE proceeds to select the new applet by invoking the `Applet.select` method on the respective (matching) applet. If the applet returns `true`, the actual SELECT APDU command is passed to the newly selected applet by invoking the `Applet.process` method.

The *getApplet* function returns the location of the applet whose AID is encoded by command *cmd*:

$$A.getApplet(cmd) = \begin{cases} A.appTable(aid) & \text{if } aid \in dom(A.appTable) \\ \texttt{null} & \text{otherwise} \end{cases}$$
$$\text{where } (aid \in \mathsf{AID}) \wedge (n = aid.length = cmd.Lc) \wedge (\forall 0 \leq i < n.aid[i] = cmd.data[i])$$

If the CAD received an APDU select command then the applet identified by the AID specified by the command (*getApplet* function) must be selected. This is done by first invoking the `select` method of the designated applet and then passing the command as argument to the `process` method. If an applet is currently selected then it must be deselected first by invoking the `deselect` method and clearing the `CLEAR_ON_DESELECT` transient objects owned by the applet being deselected.

If the applet's `select` method throws an exception or returns false , then the JCRE state is set so that no applet is selected. The JCRE responds to the SELECT command with status code 0x6999 (SW_APPLET_SELECT_FAILED).

If no applet is found that matches the AID specified by the SELECT command then the command must be processed by the currently selected applet (`process` method). If no applet is currently selected then JCRE send a reply message with status word 0x6999.

The select operation encloses a three-state transition system. The states that make up the select operation are: `Select(start)`, `Select(deselect)` and `Select(select)`. The invocation of the `process` method on the selected method is considered as part of the execution of the card with state $\texttt{Process}(p)$.

The semantics of the select operation is defined by the $card \xrightarrow{cmd} card'$ function:

- State `Select(start)`:

$$\begin{array}{l}
\langle \texttt{Select(start)}, cad, J, vm, H\rangle \xrightarrow{cmd} card' \Leftrightarrow \\
\quad (cmd = cad.\mathrm{message}) \wedge (A = J.appletCmp) \wedge (loc = A.getApplet(cmd) \wedge \\
\quad (loc \neq \perp) \wedge (A.current \neq \perp) \ \Rightarrow \\
\qquad (loc' = A.current) \wedge (o = H(loc')) \wedge \\
\qquad (m = methodIDlookup(\texttt{Applet.deselect}.id, o.class) \wedge \\
\qquad (V(0) = loc') \wedge (V.length = m.maxLocalVariableArraySize) \wedge \\
\qquad vm.config = \texttt{Run}(\langle o.own, m, 0, V, \epsilon\rangle) \wedge \langle vm', cad', J', H'\rangle = vm.eval(cad, J, H) \wedge \\
\qquad card' = \langle \texttt{Select(deselect)}, cad', J', vm', H'\rangle \\
\quad (loc \neq \perp) \wedge (A.current = \perp) \ \Rightarrow \\
\qquad (o = H(loc)) \wedge (m = methodIDlookup(\texttt{Applet.select}.id, o.class) \wedge \\
\qquad (V(0) = loc) \wedge (V.length = m.maxLocalVariableArraySize) \wedge \\
\qquad vm.config = \texttt{Run}(\langle o.own, m, 0, V, \epsilon\rangle) \wedge \langle vm', cad', J', H'\rangle = vm.eval(cad, J, H) \wedge \\
\qquad card' = \langle\ \texttt{Select(select)}, cad', J', vm', H'\rangle \\
\quad (loc = \perp) \wedge (A.current \neq \perp) \ \Rightarrow \\
\qquad \langle \texttt{Process(start)}, cad, J, vm, H\rangle \xrightarrow{cmd} card' \\
\quad (loc = \perp) \wedge (A.current = \perp) \ \Rightarrow \\
\qquad (rsp \in \mathsf{ResponseAPDU}) \wedge (rsp.data = 0) \wedge (rsp.SW1{::}rsp.SW2 = 0x6999) \\
\qquad cad' = cad[msg \mapsto rsp] \wedge card' = \langle \texttt{Reply}, cad', J, vm, H\rangle
\end{array} \tag{4}$$

- State `Select(deselect)`:

$$
\begin{array}{l}
\langle \texttt{Select(deselect)}, cad, J, vm, H\rangle \stackrel{cmd}{\longrightarrow} card' \Leftrightarrow \\
\quad (vm.config = \texttt{Run}(F^{+}) \ \Rightarrow \\
\qquad \langle vm', cad', J', H'\rangle = vm.eval(cad, J, H) \wedge (card' = \langle \texttt{Select(deselect)}, cad', J', vm', H'\rangle) \\
\quad (vm.config = \texttt{Ret}()) \vee (vm.config = \texttt{Exc}(loc)) \ \Rightarrow \\
\qquad H' = clearTransient(H, \texttt{CLEAR\_ON\_DESELECT}) \wedge A' = J.appletCmp[current \mapsto \bot] \wedge \\
\qquad J' = J[appletCmp \mapsto A'] \wedge (card' = \langle \texttt{Select(start)}, cad, J', vm, H'\rangle)
\end{array}
\tag{5}
$$

- State `Select(select)`:

$$
\begin{array}{l}
\langle \texttt{Select(deselect)}, cad, J, vm, H\rangle \stackrel{cmd}{\longrightarrow} card' \Leftrightarrow \\
\quad (vm.config = \texttt{Run}(F^{+}) \ \Rightarrow \\
\qquad\quad \langle vm', cad', J', H'\rangle = vm.eval(cad, J, H) \wedge (card' = \langle \texttt{Select(deselect)}, cad', J', vm', H'\rangle) \\
\qquad (vm.config = \texttt{Ret}(true)) \ \Rightarrow \quad \langle \texttt{Process(start)}, cad, J, vm, H\rangle \stackrel{cmd}{\longrightarrow} card' \Leftrightarrow \\
\qquad (vm.config = \texttt{Exc}(loc)) \vee (vm.config = \texttt{Ret}(false)) \ \Rightarrow \\
\qquad\quad (rsp \in \mathsf{ResponseAPDU}) \wedge (rsp.data = 0) \wedge (rsp.SW1{::}rsp.SW2 = 0x6999) \\
\qquad\quad cad' = cad[msg \mapsto rsp] \wedge card' = \langle \texttt{Reply}, cad', J, vm, H\rangle
\end{array}
\tag{6}
$$

3.2.2 Processing a non-SELECT APDU

A non-SELECT APDU command is processed by invoking the `process` method of the currently selected applet. The JCRE zeroes out the APDU buffer before receiving the new APDU command from the CAD. Then, it copies the five header bytes of the APDU command to the APDU buffer and passes the APDU object as parameter to the `process` method.

This process is also applied to SELECT APDUs after the requested applet has been successfuly selected or when the requested applet AID is not present (registered) in the card. In the latter case, the SELECT command is passed to the currently selected applet, if any.

Non-SELECT APDU processing is divided into two actions which determine two states: `Process(start)` and `Process(process)`. The operations associated to each state are defined by the $card \stackrel{cmd}{\longrightarrow} card'$ function, as follows:

- `Process(start)` denotes the state of the card before invoking the `process` method. The JCRE prepares the APDU object for processing - as defined by the *receiveHeader* function in section 10 - and sets up the JCVM configuration for execution of the respective `process` method.

$$
\begin{array}{l}
\langle \texttt{Process(start)}, cad, J, vm, H\rangle \stackrel{cmd}{\longrightarrow} card' \Leftrightarrow \\
\quad (A = J.appletCmp) \wedge (loc = A.current) \wedge \\
\quad (loc \neq \bot) \ \Rightarrow \\
\qquad (H' = J.APDUCmp.receiveHeader(cad, H)) \wedge (o = H(loc)) \wedge \\
\qquad (m = methodIDlookup(\texttt{Applet.process}.id, o.class) \wedge \\
\qquad (V(0) = loc) \wedge (V.length = m.maxLocalVariableArraySize) \wedge \\
\qquad vm' = vm[config \mapsto \texttt{Run}(\langle o.own, m, 0, V, A.apduLoc\rangle)] \wedge \\
\qquad (card' = \langle \texttt{Process(process)}, cad, J, vm', H'\rangle) \\
\quad (loc = \bot) \ \Rightarrow \\
\qquad (rsp \in \mathsf{ResponseAPDU}) \wedge (rsp.data = 0) \wedge (rsp.SW1{::}rsp.SW2 = 0x6999) \\
\qquad cad' = cad[msg \mapsto rsp] \wedge card' = \langle \texttt{Reply}, cad', J, vm, H\rangle
\end{array}
\tag{7}
$$

The resulting state must be `Process(process)`.

- The step associated to the `Process(process)` state executes one step of the JCVM, if the VM configuration is a running configuration. Upon normal return from the `process` method, the JCRE copies the ISO 7816-4 defined success status (0x9000) to the CAD (response message) and prepares the CAD for sending the response to the host (state `Reply`). If this method throws an `ISOException`, the JCRE copies the associated reason code as the response status instead.

$$
\begin{array}{l}
\langle \texttt{Process(process)}, cad, J, vm, H\rangle \stackrel{cmd}{\longrightarrow} card' \Leftrightarrow \\
\quad (vm.config = \texttt{Run}((F^{+})) \Rightarrow \\
\quad\quad \langle vm', cad', J', H'\rangle = vm.eval(cad, J, H) \wedge (card' = \langle \texttt{Process(process)}, cad', J', vm', H'\rangle) \\
\quad (vm.config = \texttt{Ret}()) \Rightarrow \\
\quad\quad (rsp.SW1{::}rsp.SW2 = 0x9000) \wedge cad' = cad[msg \mapsto rsp] \wedge \\
\quad\quad H' = J.APDUCmp.clearAPDU(H) \wedge card' = \langle \texttt{Reply}, cad', J, vm, H'\rangle \\
\quad (vm.config = \texttt{Exc}(loc')) \Rightarrow \\
\quad\quad (rsp.SW1{::}rsp.SW2 = H(loc').reason) \wedge cad' = cad[msg \mapsto rsp] \wedge \\
\quad\quad H' = J.APDUCmp.clearAPDU(H) \wedge card' = \langle \texttt{Reply}, cad', J, vm, H'\rangle \\
\text{where } (rsp \in \mathsf{ResponseAPDU}) \wedge (rsp.data = 0)
\end{array}
\tag{8}
$$

3.3 Replying to the host

Every APDU command sent by the host must be matched by a response APDU message sent by the card. The JCRE signals that it has finished processing the APDU command by setting the card state to `Reply`. At this point, the response APDU must be ready and must contain all the neccesary information in its fields: the data part - if any - filled by the applet's code and the status code generated by the JCRE. The reply operation consists on "copying" the response APDU from the CAD to the host and setting the card state to `Ready`, as defined by the respective rule in Figure 1:

$$
\frac{\begin{array}{c}(ev = \mathbf{NoEvent})\wedge \\ (cad' = cad.sendHost) \wedge \\ (card' = \langle \texttt{Ready}, cad', J, vm, H\rangle)\end{array}}{\langle \texttt{Reply}, cad, J, vm, H\rangle \longrightarrow^{ev} card'}
\tag{9}
$$

4 The Java Card API and the JCRE element

The Java Card API provides an interface that can be used by programmers to include the Java Card specific features into their applications. Without these methods, Java Card applets would be circumscribed to the functionality defined by the JCVM language subset they are written in.

The Java Card API consists of four packages: `java.lang`, `javacard.framework`, `javacard.security` and `javacardx.crypto`. The last two packages are mostly made of interfaces that declare the methods that should be used by the applet developers when dealing with security and cryptographic protocols. They have no direct relation with the internal state of the card and are, therefore, of little interest for the specification of the Java Card semantics. We shall make no further references to any of these two packages.

On the other hand, the `java.lang` and `javacard.framework` packages are an integral part of the Java Card system. The `java.lang` package provides classes that are fundamental to the design of the Java Card technology subset of the Java programming language, such as the `Object` and `Exception` classes. The `javacard.framework` package provides framework of classes and interfaces for the core functionality of a Java Card applet, such as the handling of the APDU buffer or the use of transactions.

Java Card API methods can be implemented in two ways. Some methods are heavily dependant on the implementation of the underlying JCRE (modelled by the JCRE component) and are consequently partially or completely defined as native methods. Other methods (non-native) are implemented using JCVM bytecodes, simultaneously providing a library-based implementation of certain JCRE features. Therefore, a complete specification of the `javacard.framework` package is closely related to the specification of the JCRE. We have chosen to subordinate the specification of the core API to the specification of the JCRE by expressing it in terms of the components of the JCRE. For example, methods declared in the `APDU` class are mainly defined in terms of the elements defined by the APDUCmp domain 10. Thus, the specification of the JCRE components provides the basic framework for the specification of the core API.

4.1 API Notation

API methods are specified as a special case of virtual and static method invocations, based on the semantics of the JCVM given in Secsafe-ICSTM-001 [3]. The API specification shares the framework defined for the JCVM and, consequently, re-uses the object model (ownership, transient data) and semantic machinery (states) used in the formalisation of the JCVM. Furthermore, a bytecode level specification allows us to model the actions performed by the API method calls in more detail. Most of the actions performed by the API methods affect the JCRE component which includes elements used to model the management of transactions (commit buffer), the pre-allocation of exception objects, applet execution and communication with the host via the APDU buffer.

The execution of API methods is defined by the functions[1]:

$$\begin{array}{ll} -,- \stackrel{a}{\Rightarrow} - & \in \mathit{Config} \times \mathsf{Method} \rightarrow \mathit{Config} \\ -,- \stackrel{a}{\Rightarrow}_{\mathrm{st}} - & \in \mathit{Config} \times \mathsf{Method} \rightarrow \mathit{Config} \end{array}$$

which take a configuration and a method and return the configuration that results from the execution of the method on that particular configuration. The first function corresponds to the execution of non-static functions while the latter defines the execution of static-methods.

Thus, the rule corresponding to the invocation of a non-static methods from the JCVM looks like:

$$\frac{\begin{array}{l} (I = \texttt{invokedefinite}\ \ m' \ \wedge\ \neg m'.\mathit{isStatic}) \vee I = \texttt{invokevirtual}\ \ m'' \\ (\tau_i)_1^n \rightarrow \tau_r = m'.\mathit{type} \wedge |v^*| = n \wedge \mathit{checkNull}(\mathit{loc}) \\ \left(\begin{array}{l} I = \texttt{invokevirtual}\ \ m'' \wedge o = H(\mathit{loc}) \\ m' = \left\{ \begin{array}{ll} \mathit{methodLookup}(m'', \texttt{Object}) & o \in \mathsf{ArrayObject} \\ \mathit{methodLookup}(m'', o.\mathit{class}) & \text{otherwise} \end{array} \right. \end{array} \right) \\ m'.\mathit{isNative} \wedge \langle (J,H,K), \mathit{SF}{::}\langle \mathit{own}, m, \mathit{pc}, V, S{::}\mathit{loc}{::}v^* \rangle \rangle, m' \stackrel{a}{\Rightarrow} \langle G', \mathit{SF}' \rangle \end{array}}{\langle (J,H,K), \mathit{SF}{::}\langle \mathit{own}, m, \mathit{pc}, V, S{::}\mathit{loc}{::}v^* \rangle \rangle \ \Rightarrow\ \langle G', \mathit{SF}' \rangle}$$

Similarly, the invocation of a static method from the JCVM is defined by:

$$\frac{\begin{array}{l} I = \texttt{invokedefinite}\ \ m' \ \wedge\ m'.\mathit{isStatic} \\ m'.\mathit{isNative} \wedge \langle (J,H,K), \mathit{SF}{::}\langle \mathit{own}, m, \mathit{pc}, V, S{::}\mathit{loc}{::}v^* \rangle \rangle, m' \stackrel{a}{\Rightarrow}_{\mathrm{st}} \langle G', \mathit{SF}' \rangle \end{array}}{\langle (J,H,K), \mathit{SF}{::}\langle \mathit{own}, m, \mathit{pc}, V, S{::}\mathit{loc}{::}v^* \rangle \rangle \ \Rightarrow\ \langle G', \mathit{SF}' \rangle}$$

5 Basic classes

The `java.lang` package contains the definition of the classes that form the core of the Java Card class hierarchy. The list of classes defined in the `java.lang` package is listed in figure 2. It provides the definition of the `Object` class, root of the Java Card class hierarchy, and `Throwable` class, superclass of all exceptions and errors.

[1] The notation used here for the specification of API methods use the definitions used in Secsafe-ICSTM-001, which includes K as part of the configuration and assunes the CAD is part of J. This is just a difference in notation that has to be updated

- `Object`
  - `Throwable`
    - `Exception`
      - `RuntimeException`
        - `ArithmeticException`
        - `ArrayStoreException`
        - `ClassCastException`
        - `IndexOutOfBoundsException`
          - `ArrayIndexOutOfBoundsException`
        - `NegativeArraySizeException`
        - `NullPointerException`
        - `SecurityException`

Figure 2: `java.lang` package class hierarchy

5.1 The `Object` class

The class `Object` is the root of the Java Card class hierarchy. Every class has `Object` as a superclass.

- `public Object()`

$$\frac{m' = \texttt{javacard.lang.Object.<init>()}}{\langle (J,K,H), SF{::}\langle own, m, pc, V, S{::}loc\rangle\rangle, m' \overset{a}{\Rightarrow} \langle (J,K,H), SF{::}\langle own, m, pc+1, V, S\rangle\rangle}$$

- `public boolean equals(Object)`

$$\frac{\begin{array}{c} m' = \texttt{javacard.lang.Object.equals(Object)} \\ v = \begin{cases} 1 & loc = V[0] \\ 0 & loc \neq V[0] \end{cases} \end{array}}{\langle (J,K,H), SF{::}\langle own, m, pc, V, S{::}loc\rangle\rangle, m' \overset{a}{\Rightarrow} \langle (J,K,H), SF{::}\langle own, m, pc+1, V, S{::}v\rangle\rangle}$$

5.2 Java Runtime Exceptions

The `java.lang` package contains the definition of the `java.lang.RuntimeException` class, superclass of all unchecked exceptions. It also defines the set of pre-defined runtime exceptions used by the Java Card Virtual Machine to report (recoverable) unexpected runtime problems. These exception classes are[2]:

$$\begin{array}{lcl} \mathsf{JRuntimeExceptions} & = & \{ \texttt{RuntimeException}, \texttt{ArithmeticException}, \texttt{ArrayStoreException}, \\ & & \texttt{ClassCastException}, \texttt{IndexOutOfBoundsException}, \\ & & \texttt{ArrayIndexOutOfBoundsException}, \texttt{NegativeArraySizeException}, \\ & & \texttt{NullPointerException}, \texttt{SecurityException} \} \end{array}$$

A `JCRE` owned instance of a runtime exception class is thrown by the Java Card Virtual Machine to indicate a runtime problem. These instances are defined as temporary JCRE Entry Point Objects to allow access from any applet context.

- Constructors:

$$\frac{m' = \texttt{java.lang.}e\texttt{.<init>()} \wedge e \in \mathsf{JRuntimeExceptions}}{\langle (J,K,H), SF{::}\langle own, m, pc, V, S{::}loc\rangle\rangle, m' \overset{a}{\Rightarrow} \langle (J,K,H), SF{::}\langle own, m, pc+1, V, S\rangle\rangle}$$

The semantics of throwing and catching an exception are defined in [3].

[2] The fully qualified names are of the form `java.lang.`*name*

6 Exceptions

6.1 Java Card Exceptions

The Java Card API defines a series of exception classes that are used to report errors during the execution of the API methods. These classes - which we group as $\mathsf{JCExceptions}$ - are :

$$\begin{aligned}
\mathsf{JCExceptions} &= \mathsf{JCCheckedExceptions} \cup \mathsf{JCUncheckedExceptions} \\
\mathsf{JCCheckedExceptions} &= \{\texttt{CardRuntimeException}, \texttt{APDUException}, \texttt{ISOException}, \\
&\quad \texttt{PINException}, \texttt{SystemException}, \texttt{TransactionException} \\
&\quad \texttt{javacardx.crypto.CryptoException}\} \\
\mathsf{JCUncheckedExceptions} &= \{\texttt{CardException}, \texttt{UserException}\}
\end{aligned}$$

The $\mathsf{JCExceptions}$ classes are equipped with a constructor and the `getReason`, `setReason` and `throwIt` methods. An instance of any of these classes is associated with a reason code which can be obtained using the following interface function:

$$\begin{aligned}
&\mathit{reason} \in \mathsf{JCExceptionInstance} \to \mathsf{ShortValue} \\
&\mathsf{JCExceptionInstance} = \{o \mid o \in \mathsf{ClassInst} \wedge o.\mathit{class} \in \mathsf{JCExceptions}\}
\end{aligned}$$

6.2 Exceptions and the `JCRE`

Being memory usage an important concern, the creation of unique runtime exception objects per class is enforced. JCRE-owned Java Runtime and Java Card exception class instances are allocated and kept by the JCRE so they can later be used by the JCVM (Java runtime exceptions) or by the user (Java Card exceptions) via the `thowIt` method.

We introduce the *eInst* element, a function that maps a (Java runtime and Java Card) exception class to the location of the respective (unique) exception object. We have:

$$\mathit{eInst} \in \mathsf{ExceptionInstances} = (\mathsf{JRuntimeExceptions} \cup \mathsf{JCExceptions}) \mapsto \mathsf{ReferenceValue}$$

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:

$$H \models \mathit{eInst} \Leftrightarrow \forall cl \in \mathit{dom}(\mathit{eInst}), \mathit{loc} = \mathit{eInst}(cl) : \mathit{loc} \in \mathit{dom}(H) \wedge \begin{cases} H(\mathit{loc}).\mathit{class} = cl \\ H(\mathit{loc}).\mathit{owner} = \texttt{JCRE} \\ H(\mathit{loc}).\mathit{entryPoint} = \mathit{tEP} \end{cases}$$

The *eInst* mapping is maintained by the *getException* function, which returns the location of the JCRE owned instance of the RuntimeException class passed as argument.

$$\begin{aligned}
&\mathit{getException} \in \mathsf{ExceptionCmp} \to (\mathsf{Heap} \times \mathsf{Class}) \to (\mathsf{ExceptionCmp} \times \mathsf{ReferenceValue}) \\
&E.\mathit{getException}(H, cl) = \begin{cases} (E, H, E.\mathit{eInst}(cl)) & \text{if } cl \in \mathit{dom}(E.\mathit{eInst}) \\ (E.\mathit{eInst}[cl \mapsto \mathit{loc}], H', \mathit{loc}) & \text{otherwise} \\ \quad \text{where } (H', \mathit{loc}) = \mathit{newObject}(\texttt{JCRE}, cl, \mathit{tEP}, H) & \end{cases}
\end{aligned}$$

The *getException* function considers the case where exception instances have to be allocated on demand. If all exception objects have been pre-allocated then *getException* behaves as a mere lookup function.

In addition to the conditions stated in $H \models \mathit{eInst}$ and *getException*, instances of the Java Card exception classes defined in 6.1 are associated with a *reason* code. Given reason code r, $E.\mathit{getJCException}(H, e, r)$ returns the unique instance of class e kept by the `JCRE` with the *reason* state variable set to r.

$$E.getJCException(H, e, r) = (E', H', loc) \Leftrightarrow \begin{array}{l} E.getException(H, cl) = (E', H'', loc) \\ H' = H''[loc \mapsto o] \\ o = H''(loc)[reason \mapsto r] \end{array}$$

The `throwit` method is defined by:

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.}e\texttt{.throwIt(short)} \\ e \in \mathsf{JCCheckedExceptions} \wedge F = \langle own, m, pc, V, S{::}r\rangle \\ (E', H', loc_e) = E.getJCException(H, e, r) \\ E = J.exception \wedge J' = J[exception \mapsto E'] \\ SF' = catchException(SF{::}F, H', loc_e) \end{array}}{\langle (J, K, H), SF{::}F\rangle, m' \overset{a}{\Rightarrow}_{\mathrm{st}} \langle (J', K, H'), SF'\rangle}$$

A detailed description of the to $\mathsf{JCExceptions}$ class methods can be found in appendix A.

7 Appplets and AIDs

An Application IDentifier (AID) is a sequence of bytes used to uniquely identify packages and applets. We use the AID domain to denote the set of valid AID arrays. ISO 7816-5 mandates that AIDs are at least 5 bytes long and do not exceed 16 bytes in length.

$$\mathsf{AID} = \{aid \mid aid \in [\mathsf{byte}] \wedge 5 \leq aid.length \leq 16\}$$

The first five bytes of an AID correspond to the National Registered Application provider identifier (RID) and identify the applet provider. The *RIDEquals* is a predicate that verifies that two AIDs have the same RID.

$$\begin{array}{l} RIDEquals \in \mathsf{AID} \times \mathsf{AID} \rightarrow \mathsf{boolean} \\ \quad RIDEquals(aid, aid') = \forall i, 0 \leq i < 5, aid[i] = aid'[i] \end{array}$$

The second portion of an AID - 0 to 11 bytes long - is known as the Propietary Identifier Extension (PIX) and is used to uniquely identify a package or applet that belong to same provider.

Package and default applet AIDs are kept in the CAP file and are supplied to the converter when the CAP file is generated. Other applet AIDs are supplied by the user using the APDU buffer (section 10) during the installation process.

In this section we describe the operations and domains that deal directly with applet and AID information. Applet operations performed by the JCRE are summarised by the specification of the Applet component, listed in Figure 3.

7.1 Applet Table

All applet instances must be registered with the JCRE before they can be selected and set to run. Applet registration is performed during the installation process and consists of assigning a unique AID to the new applet. The new applet can be assigned the default applet AID or an AID supplied during the installation process using the APDU buffer (section 10). We model the process of applet registration by defining a special structure to keep track of all registered applets, the applet table:

$$\begin{array}{rcl} appTable & \in & \mathsf{AppletCmp} \mapsto \mathsf{AppletTable} \\ current & \in & \mathsf{AppletCmp} \mapsto \mathsf{Location}_{\perp} \\ applets & \in & \mathsf{AppletTable} = \mathsf{AID} \xrightarrow{1-1} \mathsf{Location} \end{array}$$

The applet table is a one-to-one mapping from elements of the AID domain to locations and can be obtained from the JCRE applet component using the *appTable* function. The object locations registered in an applet

$$
\begin{array}{rcl}
\hline
appTable & \in & \mathsf{AppletCmp} \mapsto \mathsf{AppletTable} \\
current & \in & \mathsf{AppletCmp} \mapsto \mathsf{Location}_{\perp} \\
aidTable & \in & \mathsf{AppletCmp} \rightarrow \mathsf{AIDTable} \\
getAID & \in & \mathsf{AppletCmp} \rightarrow \mathsf{GetAID} \\
\hline
\mathsf{AppletTable} & = & \mathsf{AID} \xrightarrow{1-1} \mathsf{Location} \\
\mathsf{AIDTable} & = & \mathsf{AID} \mapsto \mathsf{Location} \\
\mathsf{GetAID} & = & (\mathsf{Heap} \times \mathsf{AID}) \rightarrow (\mathsf{AppletCmp} \times H \times \mathsf{Location}) \\
\hline
\end{array}
$$

Figure 3: AppletCmp : The Applet Component

table must correspond to applet references. We say that applet table *applets* is well-formed with respect to heap H if:

$$H \models applets \Leftrightarrow \forall aid \in dom(applets).\ H \vdash applets(aid)\ :\preceq \texttt{Applet}$$

The applet component keeps track of the current applet with the *current* field. If no applet is currently select, *current* returns $\perp$.

7.2 The AID class

AID information is usually manipulated by an applet using instances of the `AID` class. The `AID` class encapsulates the bytes that define an AID and contains methods to extract and compare this information. We model the presence of the AID bytes in an `AID` object by assigning an element of the AID domain to instances of the `AID` class or any of its subclasses. This information can be extracted from an object using the *theAID* interface function:

$$
\begin{array}{l}
\mathsf{AIDInstances} = \{o \in \mathsf{ClassInst} \mid (o \in \mathsf{ClassInst}) \wedge o.class \preceq \texttt{AID}\} \\
theAID \in \mathsf{AIDInstances} \mapsto \mathsf{AID}
\end{array}
$$

The AID information encapsulated by an `AID` class instance is provided by the program using the class constructor. The constructor takes three arguments: the location of the byte array $bArray$ containing the AID bytes, the start of the AID bytes in $bArray$, and the length of the AID bytes in $bArray$. The rule for the `AID` constructor is specified below:

$$
\frac{
\begin{array}{l}
m' = \texttt{javacard.framework.AID.<init>(byte[], short, byte)} \\
(loc_a \neq \texttt{null}) \wedge (0 \leq offset \leq H(loc_a).length - l) \\
(5 \leq l \leq 16) \wedge (aid.length = l) \wedge \forall i, 0 \leq i < l, aid[i] = H(loc_a).values[offset + i] \\
o = H(loc)[theAID \mapsto aid] \wedge H' = H[loc \mapsto o]
\end{array}
}{
\langle\!\langle (J,K,H), SF{::}\langle own, m, pc, V, S{::}loc{::}loc_a{::}offset{::}l\rangle\rangle\!\rangle, m' \overset{a}{\Rightarrow} \langle\!\langle (J,K,H), SF{::}\langle own, m, pc+1, V, S\rangle\rangle\!\rangle
}
$$

Once the AID bytes have been assigned to the `AID` object, they can be extracted using `getBytes` method or compared using the remainder methods but not modified. The specification of all the `API` class methods can be found in appendix B.

7.3 Applet AID class instances

The JCRE creates `AID` class instances to identify and manage every applet on the card. Applets need not create instances of this class. An applet may request and use the JCRE owned instances to identify itself and other applet intances[5].

The JCRE owned class instances of the applet registered in the card are kept in an AID table, a one-to-one mapping from elements of the AID domain to the locations of the respective `AID` objects, and can be accessed using the *aidTable* interface function defined below:

$$\begin{array}{rcl} \mathit{aidTable} & \in & \mathsf{AppletCmp} \to \mathsf{AIDTable} \\ \mathsf{AIDTable} & = & \mathsf{AID} \mapsto \mathsf{Location} \end{array}$$

All locations kept in an *aidTable* must be references to JCRE owned instances of the `AID` class. Furthermore, they are temporary JCRE Entry Point Objects and can be accessed from any context and their references, stored and re-used. We say that *aidTable* is well-formed with respect to heap H if:

$$H \models \mathit{aidTable} \Leftrightarrow \forall \mathit{aid} \in \mathit{dom}(\mathit{aidTable}). \begin{cases} \mathit{aidTable}(\mathit{aid}) \in \mathit{dom}(H) \wedge o = H(\mathit{aidTable}(\mathit{aid})) \\ H \vdash o \ : \ \mathtt{AID} \\ \mathit{aid}.\mathit{equals}(o.\mathit{theAID}) \\ o.\mathit{isObject}(\mathit{JCRE}, \mathtt{AID}, \mathit{tEP}) \end{cases}$$

The notion of well-formedness is extended to the pair $(\mathit{appTable}, \mathit{aidTable})$ as follows:

$$H \models (\mathit{appTable}, \mathit{aidTable}) \Leftrightarrow H \models \mathit{appTable} \wedge H \models \mathit{aidTable} \wedge \mathit{dom}(\mathit{aidTable}) = \mathit{dom}(\mathit{appTable})$$

We only require that, besides both elements being well-formed, all AIDs present in the AID table correspond to registered applets i.e. they belong to the domain of AIDs of the applet table. They need not be equal because the specification allows to create AID objects on demand, whenever the program requests for an specific AID of a registered applet. The specification is complete by adding the definition of the *getAID* function:

$$\mathit{getAID} : \mathsf{AppletCmp} \to (\mathsf{Heap} \times \mathsf{AID}) \to (\mathsf{AppletCmp} \times H \times \mathsf{Location})$$

$$A.\mathit{getAID}(H, \mathit{aid}) = \begin{cases} (A, H, \mathtt{null}) & \mathit{aid} \notin \mathit{dom}(A.\mathit{appTable}) \\ (A, H, \mathit{loc}) & \mathit{aid} \in \mathit{dom}(A.\mathit{aidTable}) \wedge \mathit{loc} = A.\mathit{aidTable}(\mathit{aid}) \\ (A', H', \mathit{loc}) & (H'', \mathit{loc}) = \mathit{newObject}(\mathtt{JCRE}, \mathtt{AID}, \mathit{tEP}, H) \wedge \\ & o = H''(\mathit{loc})[\mathit{theAID} \mapsto \mathit{aid}] \wedge H' = H''[\mathit{loc} \mapsto o] \\ & A' = A.\mathit{aidTable}[\mathit{aid} \mapsto \mathit{loc}] \end{cases}$$

The *getAID* function creates an instance of an `AID` object if the AID specified corresponds to a registered applet that hasn't been allocated yet. The *getAID* function can be simplified to a mere lookup function by enforcing $\mathit{dom}(\mathit{aidTable}) = \mathit{dom}(\mathit{appTable})$. In that case, an AID object will be created by the JCRE everytime a new applet is registered.

7.4 API methods for Applet AID manipulation

The `JCSystem` class includes a collection of methods that control applet execution and resource management. The methods involved in the manipulation of AID and applet information are:

```
public final class JCSystem {
  public static AID getAID();
  public static AID getPreviousContextAID();
  public static AID lookupAID(byte[], short, byte);
...}
```

All three methods return the JCRE owned `AID` instances mantained by the AID table. The first two methods return the `AID` object that encapsulates the AID associated with the current and the previously active applet, respectively. The `lookupAID` returns the `AID` object that matches the AID bytes specified by the method arguments.

```
abstract public class Applet {
  protected Applet();
  public void deselect();
  Shareable getShareableInterfaceObject(AID clientID, byte parameter);
  static void install(byte[] bArray, short bOffset, byte bLength);
  public abstract void process(APDU apdu);
  protected void register();
  protected void register(byte[] bArray, short bOffset, byte bLength);
  public boolean select();
  protected boolean selectingApplet();
};
```

Figure 4: The `Applet` class

The current applet AID is determined by the AID of the owner of the current frame. The previously active applet is determined by performing a search on the VM's configuration. This search is defined by the *getPreviousContext* function:

$$\begin{array}{l} \mathit{getPreviousContext} : \mathsf{Frame}^* \times \mathsf{Owner} \to \mathsf{Owner} \\ \mathit{getPreviousContext}(\epsilon, \mathit{own}') = \mathit{JCRE} \\ \mathit{getPreviousContext}(\mathit{SF}{::}\langle \mathit{own}, m, \mathit{pc}, V, S\rangle, \mathit{own}') = \left\{ \begin{array}{ll} \mathit{own} & \text{if } \mathit{own.aid} \neq \mathit{own.'aid} \\ \mathit{getPreviousContext}(\mathit{SF}, \mathit{own}) & \text{otherwise} \end{array} \right. \end{array}$$

The rule corresponding to the `JCSystem.getPreviousContextAID()` method is shown below. First, the call frame stack and the current owner information are used by the *getPreviousContext* to determine the AID of the applet that called the current applet, if any. The `AID` instance associated with the applet found in the previous step is obtained from the JCRE and returned (pushed on top of the stack). The `null` reference is returned if the caller was the JCRE.

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.getPreviousContextAID()} \\ \mathit{own}' = \mathit{getPreviousContext}(\mathit{SF}{::}\langle \mathit{own}, m, \mathit{pc}, V, S\rangle, \mathit{own}) \\ (J', H', \mathit{loc}) = \left\{ \begin{array}{ll} (J, H, \texttt{null}) & \mathit{own}' = \texttt{JCRE} \\ J.\mathit{getAID}(H, \mathit{own.aid}) & \text{otherwise} \end{array} \right. \end{array}}{\langle (J, K, H), \mathit{SF}{::}\langle \mathit{own}, m, \mathit{pc}, V, S\rangle\rangle, m' \stackrel{a}{\Rightarrow}_{\mathrm{st}} \langle (J', K, H'), \mathit{SF}{::}\langle \mathit{own}, m, \mathit{pc}+1, V, S{::}\mathit{loc}\rangle\rangle}$$

A detailed description of the methods can be found in Appendix C.1

7.5 The Java Card Applet Lifetime

A Java Card applet's lifetime begins at the point it has been correctly loaded into card memory, linked, and otherwise prepared for execution. The state of the applet will change upon execution of the methods declared in the `Applet` class (Figure 4). Applet registration is performed by the `register` method. The JCRE interacts with the applet via the `Applet` class public methods `install`, `select`, `deselect`, and `process`. Most of these methods must be implemented by the applet.

8 Atomicity and Transactions

Atomicity defines how the card handles the contents of updated *persistent data* after a stop, failure, or fatal exception during the update of a single or group of data elements (object field, class field and array component).

The Java Card platform guarantees that any update to a single persistent data element will be atomic. That is, if the smart card loses power during the update of a data element, its content is restored to its previous value.

An applet might need to ensure the atomic update of several different data elements. For such cases, the Java Card platform supports a transactional model in which an applet can group a set of updates into a single transaction. The card data is restored to its original pre-transaction state if the transaction does not complete normally.

Finally, the Java Card platform also provides methods that guarantee atomicity for block updates of array elements (`Util` class).

8.1 Transactions

A transaction is a logical set of updates of persistent data. The Java Card platform supports atomic transactions. The Java Card API methods - all part of the `JCSystem` class - that implement and manage atomic transactions, and the elements of the Transaction component used to specify them, are listed in Figure 1.

An applet designates the beginning and (normal) termination of a transaction with calls to the `beginTransaction` and `commitTransaction` methods, respectively. All updates to persistent data elements performed during the transaction are conditional until a call to the `commitTransaction` method commits all conditional updates to persistent storage.

A transaction can be aborted by the user with a call to the `abortTransaction` method, or by the JCRE in the event of power failure or error. In the latter case, the JCRE calls the `abortTransaction` method when it regains control. If a transaction is aborted, all conditionally updated data elements are restored to their pre-transaction values (rollback).

The *transactionDepth* interface function is used to model the current transaction nesting depth level, where a value of 0 corresponds to the state where no transaction is in progress. The value of *transactionDepth* is modified by the calls to `beginTransaction`, where it is incremented by one, and by calls to `commitTransaction` and `abortTransaction`, where it is decremented by one. For example, the result of applying the `beginTransaction` method, as described by the rule shown above,

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.beginTransaction()} \\ T = J.\mathit{transactionCmp} \wedge T.\mathit{transactionDepth} = 0 \\ T' = T[\mathit{transactionDepth} \mapsto 1] \wedge J' = J[\mathit{transactionCmp} \mapsto T'] \end{array}}{\langle\!\langle (J, K, H), \mathit{SF}{::}\langle \mathit{own}, m, \mathit{pc}, V, S\rangle\!\rangle, m' \overset{a}{\Rightarrow}_{\mathrm{st}} \langle\!\langle (J', K, H), \mathit{SF}{::}\langle \mathit{own}, m, \mathit{pc}+1, V, S\rangle\!\rangle}$$

is to increment the value of *transactionDepth* by one, provided that it was initially zero. According to the current specification, only one transaction can be in progress at a time. Nested transactions are not allowed and, consequently, *transactionDepth* can only hold values of 0 and 1. If the `beginTransaction` is called while a transaction is already in progress (*transactionDepth* equals to 1), then a `TransactionException` is thrown. Similarly, if the `commitTransaction` or `abortTransaction` methods are called while a transaction is not in progress (*transactionDepth* equals to 0), then a `TransactionException` is thrown.

8.1.1 The Commit Buffer

The JCRE maintains a commit buffer to implement the atomicity of transactions. The commit buffer is used to guarantee the following conditions:

- Inside a transaction, a read operation must yield the value of the last update.
- If a transaction terminates successfully (a call to `commitTransaction`), a read operation to a data element updated inside the transaction must reflect its latest update.

T	$\in$	$\mathsf{TransactionCmp}$
$transactionDepth$	$\in$	$\mathsf{TransactionCmp} \rightarrow \mathsf{ByteValue}$
$maxCommitCapacity$	$\in$	$\mathsf{TransactionCmp} \rightarrow \mathsf{ShortValue}$
$commitBuffer$	$\in$	$\mathsf{TransactionCmp} \rightarrow \mathsf{CommitBuffer}$
$cb \in \mathsf{CommitBuffer}$	$=$	$\mathsf{ElementLocation} \rightarrow \mathsf{Value}$
$fieldStoreTransaction$	$\in$	$\mathsf{TransactionCmp} \rightarrow (H \times \mathsf{Location} \times \mathsf{FieldID} \times \mathsf{Value}) \rightarrow \mathsf{TransactionCmp}$
$arrayStoreTransaction$	$\in$	$\mathsf{TransactionCmp} \rightarrow (H \times loc \times \mathsf{ShortValue} \times \mathsf{Value}) \rightarrow \mathsf{TransactionCmp}$
$restoreNewObjects$	$\in$	$\mathsf{TransactionCmp} \rightarrow \mathsf{Heap} \rightarrow (\mathsf{Heap} \times \mathsf{LocalVar} \times \mathsf{OperandStack})$
$restoreUpdates$	$\in$	$\mathsf{TransactionCmp} \rightarrow (\mathsf{Heap} \times \mathsf{LocalVar} \times \mathsf{OperandStack}) \rightarrow (\mathsf{LocalVar} \times \mathsf{OperandStack})$

```
JCSystem class :
    public static void abortTransaction();
    public static void beginTransaction();
    public static void commitTransaction();
    public static short getMaxCommitCapacity();
    public static byte getTransactionDepth();
    public static short getUnusedCommitCapacity();
```

Table 1: The Transaction Component and Transaction API methods

- If a transaction aborts, a read operation to a persistent data element must yield its pre-transaction value.

Updates to persistent data elements performed during a transaction are conditional. Conditional updates can be "undone" and rolled back to their pre-transaction values if the transaction is aborted, or committed when the transaction ends successfully.

The commit buffer is used to implement conditional updates. When a transaction is in progress, The JCRE journals all updates to persistent data elements into the commit buffer so that it can always guarantee the commit and rollback operations. We can mention at least two implementation approaches, a pessimistic and optimistic approach[1]. The optimistic approach considers that the normal termination of a transaction is more likely and performs updates to persistent data directly to the heap. The commit buffer is used to record the pre-transaction values of the modified data elements and used if the transaction aborts. The pessimistic approach considers that the abortion of a transaction is more likely and only writes the final values to the heap when the transaction is committed. The commit buffer works as a cache and temporarily (conditionally) records the new values of persistent data modified during the transaction.

The representation style of our specification follows the optimistic approach. The commit buffer is used to store the initial values of all persistent data elements updated during a transaction. It also keeps track of the locations of all objects allocated since the begining of a transaction. The interface functions related to the commit buffer are:

$commitBuffer$	$\in$	$\mathsf{TransactionCmp} \rightarrow \mathsf{CommitBuffer}$
$maxCommitCapacity$	$\in$	$\mathsf{TransactionCmp} \rightarrow \mathsf{ShortValue}$
$cb \in \mathsf{CommitBuffer}$:		
$oldValues$	$\in$	$\mathsf{CommitBuffer} \rightarrow \mathsf{ElementLocation} \rightarrow \mathsf{Value}$
$newLocatios$	$\in$	$\mathsf{CommitBuffer} \rightarrow \mathcal{P}(\mathsf{Location})$
$updateBuffer$	$\in$	$\mathsf{CommitBuffer} \times \mathsf{ElementLocation} \times \mathsf{Value} \rightarrow \mathsf{CommitBuffer}$
$usedCommitCapacity$	$\in$	$\mathsf{CommitBuffer} \rightarrow \mathbb{N}$

The *newLocations* returns the set of the locations of all objects allocated during the transaction. The *oldValues* function returns a mapping from data element locations (array entries or fields) to values used to store old values the first time an update to a persistent data element is reported. This operation is performed by the *updateBuffer* function, defined as follows:

$$updateBuffer(cb, eloc, v) = cb' \text{ where}$$
$$cb' = \begin{cases} cb.oldValues[eloc \mapsto v] & \text{if } eloc \notin dom(cb.oldValues) \\ cb & \text{otherwise} \end{cases}$$

The *usedCommitCapacity* function returns the number of bytes used to store the information in *commitBuffer*. It is implementation dependent and is left undefined. We show a possible implementation where the bytes required to store all element locations kept by the buffer, and the (old) values stored associated with them, are added together:

$$usedCommitCapacity(cb) = \sum_{eloc \in dom(cb)} (nbBytes(eloc) + nbBytes(cb(eloc)))$$

Commit buffer updates are trigered by the execution of the JCVM instructions that deal with field and array update. This interaction beetween the JCVM and the specification of atomic transactions is modelled by the *fieldStoreTransaction* and *arrayStoreTransaction* functions, used to ensure that the initial value of fields or array entries of non-transient objects (only transient arrays are allowed) updated inside a transaction are reported to the commit buffer. The *fieldStoreTransaction* function,

- $fieldStoreTransaction \in \mathsf{TransactionCmp} \rightarrow (H \times \mathsf{Location} \times \mathsf{FieldID} \times \mathsf{Value}) \rightarrow \mathsf{TransactionCmp}$:
  $$T.fieldStoreTransaction(H, loc, id, v) = T[commitBuffer \mapsto cb'],$$
  $$cb' = \begin{cases} updateBuffer(T.commitBuffer, (loc, id), v) & \text{if } (T.transactionDepth \neq 0) \\ T.cb & \text{otherwise} \end{cases},$$

is used by the `putStatic` and `putField` instructions to report all updates to persistent objects performed during a transaction. Similarly, the *arrayStoreTransaction* is used by the `arrayStore` method to report all updates to elements of persistent array performed during a transaction. It is defined by:

- $arrayStoreTransaction \in \mathsf{TransactionCmp} \rightarrow (H \times loc \times \mathsf{ShortValue} \times \mathsf{Value}) \in \mathsf{TransactionCmp}$:
  $$T.arrayStoreTransaction(H, loc, i, v) = T[commitBuffer \mapsto cb'],$$
  $$cb' = \begin{cases} updateBuffer(T.commitBuffer, (loc, i), v) & \text{if } (T.transactionDepth \neq 0) \wedge \\ & \quad (H(loc).transient \neq \texttt{NOT\_TRANSIENT}) \\ T.cb & \text{otherwise} \end{cases}$$

8.1.2 Abort and Transaction Failure

The atomic transaction model must ensure that all updates performed to a persistent object during a transaction are rolled-back i.e. restored to their pre-transaction state, if the transaction is aborted. This action - defined by the *restoreUpdates* function below - is performed by taking the information stored in the commit buffer and writing it back to the corresponding field or data element. The effect of this action is an updated heap and static memory.

$$T.restoreUpdates(H) = (H', K') \text{ where } old = T.commitBuffer.oldValues \wedge$$
$$H' = H[eloc_1 \mapsto old(eloc_1)] \ldots [eloc_n \mapsto old(eloc_n)] \wedge \{eloc_1, \ldots, eloc_n\} = dom(old) \backslash \mathsf{StaticElement}$$
$$K' = K[eloc'_1 \mapsto old(eloc'_1)] \ldots [eloc'_n \mapsto old(eloc'_m)] \wedge \{eloc_1, \ldots, eloc_n\} = dom(old) \cap \mathsf{StaticElement}$$

In addition to the above, all objects created during the aborted transaction must be cleared from memory, and their references set to `null`. This task is accomplished by the *restoreNew* which sets to `null` all the references in the stack and local variable array to objects instantiated from within the aborted transaction.

$$
\begin{array}{l}
T.\mathit{restoreNewObjects}(H, V, S) = (H', V', S') \text{ where} \\
\quad \mathit{locs} = T.\mathit{commitBuffer}.\mathit{newLocations} \\
\quad V.\mathit{update}(\mathit{values}) \wedge \mathit{values} = \{(i, \mathtt{null}) \mid (0 \leq i < V.\mathit{length}) \wedge V[i] \in \mathit{locs}\} \\
\quad S.\mathit{update}(\mathit{values}) \wedge \mathit{values} = \{(i, \mathtt{null}) \mid (0 \leq i < S.\mathit{length}) \wedge S[i] \in \mathit{locs}\} \\
\quad \mathit{dom}(H') = \mathit{dom}(H) \backslash \mathit{locs} \ \wedge \ \forall \mathit{loc} \in \mathit{dom}(H').\ H'(\mathit{loc}) = \mathit{clearObject}(H(\mathit{loc}), \mathit{locs})
\end{array}
$$

where

$$
\mathit{clearObject}(o, \mathit{locs}) = \begin{cases} o.\mathit{updateInstance}(\mathit{values}) & \mathit{fv} = o \in \mathsf{ClassInst} \wedge \mathit{fv} = o.\mathit{fieldValue} \\ & \mathit{values} = \{(\mathit{id}, \mathtt{null}) \mid \mathit{id} \in \mathit{dom}(\mathit{fv}) \wedge \mathit{fv}(\mathit{id}) \in \mathit{locs}\} \\ o.\mathit{updateArray}(\mathit{values}) & o \in \mathsf{ArrayObject} \wedge \\ & \mathit{values} = \{(i, \mathtt{null}) \mid (0 \leq i < a.\mathit{length}) \wedge o[i] \in \mathit{locs}\} \end{cases}
$$

The *restoreNewObjects* function is difficult to implement and many JCRE implementations may not recover heap space used by the new object instances and may lock up the card session to force tear/reset processing.

9 Transient Objects

Applets sometimes require objects that contain temporary (transient) data i.e. data that need not be persistent across CAD sessions. The Java Card technology provides methods to create transient arrays with primitive components or references to `Object`. These methods are:

In `JCSystem` class:
```
public static native byte isTransient(Object);
public static native boolean[] makeTransientBooleanArray(short, byte);
public static native byte[] makeTransientByteArray(short, byte);
public static native Object[] makeTransientObjectArray(short, byte);
public static native short[] makeTransientShortArray(short, byte);
```

Transiency information can be extracted from an object using the *transient* interface function [3]. This information is provided as a parameter to the *newArray* function when a new transient object is allocated, as in the rule below that describes the creation of transient boolean arrays:

$$
\frac{\begin{array}{l} m' = \mathtt{javacard.framework.JCSystem.makeTransientBooleanArray(short,\ byte)} \\ e \in \{\mathtt{CLEAR\_ON\_RESET}, \mathtt{CLEAR\_ON\_DESELECT}\} \\ (e = \mathtt{CLEAR\_ON\_DESELECT}) \ \Rightarrow \ (\mathit{own}.\mathit{context} = \mathit{getAppletContext}(\mathit{SF}{::}\langle\mathit{own}, m, \mathit{pc}, V, S{::}l{::}e\rangle)) \\ (H', \mathit{loc}) = \mathit{newArray}(\mathit{own}, \mathtt{bool}, l, \mathtt{false}, e, H) \end{array}}{\langle(J, K, H), \mathit{SF}{::}\langle\mathit{own}, m, \mathit{pc}, V, S{::}l{::}e\rangle\rangle, m' \ \overset{a}{\Rightarrow}_{\mathrm{st}} \ \langle(J, K, H'), \mathit{SF}{::}\langle\mathit{own}, m, \mathit{pc}+1, V, S{::}\mathit{loc}\rangle\rangle}
$$

In the rule above, the value of e determines one of the two events that causes the fields of the object to be cleared. `CLEAR_ON_DESELECT` objects are cleared whenever the applet is deselected. `CLEAR_ON_RESET` object are cleared when the card is reset. The latter are used for mantaining states across applet selections.

9.1 Clearing Transient Objects

Only the contents of the fields of an object marked as transient have a transient nature. The contents of these fields shall be cleared to the field's default value at the occurrence of certain events, as explained below. The functions involved in clearing transient objects - currently only arrays can be transient - are:

$$
\begin{array}{rcl}
\mathit{clearTransient} & \in & \mathsf{Heap} \times \mathsf{Transient} \rightarrow \mathsf{Heap} \\
\mathit{clearArray} & \in & \mathsf{ArrayObject} \rightarrow \mathsf{ArrayObject}
\end{array}
$$

The *clearTransient* traverses the heap looking for arrays that are either `CLEAR_ON_RESET` or `CLEAR_ON_DESELECT`, according to the value of argument e:

$$
\mathit{clearTransient}(H, e) = H' \;\Leftrightarrow\; \begin{cases} \mathit{dom}(H) = \mathit{dom}(H') \wedge \\ \forall \mathit{loc} \in \mathit{dom}(H), : \\ \quad H'(\mathit{loc}) = \begin{cases} \mathit{clearArray}(H(\mathit{loc})) & H(\mathit{loc}) \in \mathsf{ArrayObject} \\ & H(\mathit{loc}).\mathit{transient} = e \\ H(\mathit{loc}) & \text{otherwise} \end{cases} \end{cases}
$$

All arrays that meet the criteria are cleared. Let $a \in \mathsf{ArrayObject}$ and $\tau = a.\mathit{elementType}$. The *clearArray* function is defined by:

$$
\mathit{clearArray}(a) = \mathit{updateArray}(\{(i, \mathit{def}(\tau) \mid (0 \leq i < a.\mathit{length})\})
$$

10 The APDU and applet communication with the host

In this section we model the most important aspects of the communication between on-card applications and the host. Communication between card and host is performed using messages which conform to the Application Protocol Data Unit (APDU) format, as defined in ISO specification 7816-4. There are two kinds of APDU messages: command and response. The host triggers the communication process by sending an APDU command to the card. Commands are received and inspected by the JCRE. Depending on its type, a command may be handled entirely by the JCRE or may involve the execution of code of one of the downloaded CAP files. Once the command is processed, the JCRE sends a response APDU message to the host. This section deals with the case where a command (and possibly part of the response) is sent by the JCRE to the currently selected applet for processing. We model the most important aspects of APDU processing needed for the specification of the API methods (`APDU` class) used by an applet to read a command and send a response. The overall picture, the Java Card top-level semantics, is described in section 3

We define $\mathsf{APDUCmp}$, the part of the JCRE dedicated to APDU processing. The APDU component is extracted from the JCRE element by applying the following function:

$$
\mathit{apduCmp} \in \mathsf{JCRE} \rightarrow \mathsf{APDUCmp}
$$

The elements and functionality of $\mathsf{APDUCmp}$ are summarised in Figure 5 and explained in detail in the remaining of this section.

10.1 The CAD and APDU Messages

A smart card is ready to start communication with off-card applications after it has been inserted into a Card Acceptance Device (CAD). Host applications direct the process of communication with the smart card by sending an APDU command. Once it has been placed in the CAD, a smart card waits for the host application to send an APDU command.

APDU commands conform to the following structure:

$$
\begin{array}{rcl}
\mathsf{CommandAPDU} & = & (\mathit{CLA} : \mathsf{byte}) \times (\mathit{INS} : \mathsf{byte}) \times (\mathit{P1} : \mathsf{byte}) \times (\mathit{P2} : \mathsf{byte}) \times \\
& & (\mathit{Lc} : \mathsf{byte}) \times (\mathit{data} : [\mathsf{byte}]) \times (\mathit{Le} : \mathsf{byte})
\end{array}
$$

An APDU command is divided into two parts: header and body. The header determines the type of command and it is made of the *CLA*, *INS*, *P1* and *P2* bytes which correspond to the class of instruction, instruction

$$
\begin{array}{rcl}
\hline
locAPDU & \in & \mathsf{APDUCmp} \rightarrow \mathsf{Location} \\
protocol & \in & \mathsf{APDUCmp} \rightarrow \mathsf{Protocol} \\
NAD & \in & \mathsf{APDUCmp} \rightarrow \mathsf{byte} \\
IFSC & \in & \mathsf{APDUCmp} \rightarrow \mathsf{short} \\
inBlockSize & \in & \mathsf{APDUCmp} \rightarrow \mathsf{ShortValue} \\
outBlockSize & \in & \mathsf{APDUCmp} \rightarrow \mathsf{ShortValue} \\
\hline
\mathsf{APDUObject} & = & \mathsf{ClassInst}^{\mathtt{APDU}} \\
locBuffer & \in & \mathsf{APDUObject} \rightarrow \mathsf{Location} \\
state & \in & \mathsf{APDUObject} \rightarrow \mathsf{APDUState} \\
Le & \in & \mathsf{APDUObject} \rightarrow \mathsf{ShortValue} \\
Lc & \in & \mathsf{APDUObject} \rightarrow \mathsf{ShortValue} \\
isChaining & \in & \mathsf{APDUObject} \rightarrow \mathsf{boolean} \\
\hline
\mathsf{Protocol} & = & \{\mathtt{T0}, \mathtt{T1}\} \subseteq \mathsf{ByteValue} \\
\mathsf{APDUState} & = & \{\mathtt{READY}, \mathtt{IN/SET}, \mathtt{IN/READY}, \mathtt{IN/REC}, \mathtt{OUT/SET}, \mathtt{OUT/READY}, \mathtt{OUT/REC}\} \\
\hline
\end{array}
$$

Figure 5: The APDU Component

code, and parameters 1 and 2, respectively. The body is made of the *Lc*, *data* and *Le* fields. The *data* field contains the data to be processed by the command specified by the header. The *Lc* field specifies the length of *data* while *Le* contains the number of bytes expected by the host in the card's response.

Once the command APDU has been processed, the card sends an APDU response to the host. Thus, each command APDU is paired by a response APDU. A response APDU conforms to the following structure:

$$\mathsf{ResponseAPDU} = (data : [\mathsf{byte}]) \times (SW1 : \mathsf{byte}) \times (SW2 : \mathsf{byte})$$

It consists of an optional body (*data* field) and a mandatory trailer (*SW1* and *SW2*). The trailer consists of two fields - added by the JCRE - that together form the status word and denote the processing state of the card after executing the command APDU.

We define the CAD domain to model the transfer of APDU messages between host applications and the JCRE. An element of the CAD domain contains information about the message that is currently being processed (*message*) and the number of bytes of data (*ldata*) that have been read - if the message is a CommandAPDU - or written, if the message is a ResponseAPDU).We use `NoMsg` to model the absence of messages.

The CAD domain is made of the following elements:

$$
\begin{array}{rcl}
cad & \in & \mathsf{CAD} \\
\mathsf{MsgAPDU} & = & \mathsf{CommandAPDU} \cup \mathsf{RespondeAPDU} \cup \{\mathtt{NoMsg}\} \\
message & \in & \mathsf{CAD} \rightarrow \mathsf{MsgAPDU} \\
ldata & \in & \mathsf{CAD} \rightarrow \mathsf{short} \\
readData & \in & \mathsf{CAD} \rightarrow \mathsf{short} \rightarrow (\mathsf{CAD} \times \mathsf{ByteValue}^*) \cup \{\mathtt{I/O\text{-}error}\} \\
writeData & \in & \mathsf{CAD} \rightarrow \mathsf{ByteValue}^* \rightarrow \mathsf{CAD} \\
copyCmd & \in & \mathsf{CAD} \rightarrow (\mathsf{CommandAPDU} \rightarrow \mathsf{CAD}) \\
sendHost & \in & \mathsf{CAD} \rightarrow \mathsf{CAD}
\end{array}
$$

The *readData* function reads data from the *message* field of the CAD. It takes as argument the requested number of bytes to be read and returns, if the operation is succesful, the sequence of bytes actually read from the CAD and the new CAD state that reflects the change in *ldata*. The definition of *readData* is non-deterministic since there is no guarante on the number of bytes actually read from the CAD or on the success of the operation. The latter is denoted by an `I/O-error` return value. The complete definition is:

$$
\begin{array}{l}
\mathit{readData} \in \mathsf{CAD} \rightarrow \mathsf{short} \rightarrow (\mathsf{CAD} \times \mathsf{ByteValue}^*) \cup \{\mathtt{I/O\text{-}error}\} \\
\mathit{receiveData}(\mathit{cad}, l) = (\mathit{cad}', \mathit{seq}) \Leftrightarrow \\
\qquad (\mathit{msg} = \mathit{cad}.\mathit{message}) \wedge (\mathit{rem} = \mathit{msg}.\mathit{data}.\mathit{length} - \mathit{cad}.\mathit{ldata}) \\
\qquad (((l' = l) \wedge (l \leq \mathit{rem})) \vee ((l' = \mathit{rem}) \wedge (l > \mathit{rem}))) \\
\qquad (l' = \mathit{seq}.\mathit{length}) \wedge \forall i.\ (0 \leq i < l').\ \mathit{seq}[i] = \mathit{msg}.\mathit{data}[l + i] \\
\qquad \mathit{cad}' = \mathit{cad}[\mathit{ldata} \rightarrow \mathit{ldata} + l'] \\
\mathit{receiveData}(\mathit{cad}, l) = \mathtt{I/O\text{-}error} \Leftrightarrow \mathtt{true}
\end{array}
$$

Similarly, *writeData* takes as input a sequence of bytes and appends them to bytes in *message* The *copyCmd* is used to copy a CommandAPDU from the host to the CAD. *sendHost* is used to send the RespondeAPDU from the CAD to the host.

10.2 The APDU Object and APDU buffer

Applications do not work directly with the host. Instead, applications deal with the JCRE by reading and writing information from and into an array of bytes - called the APDU buffer. The APDU buffer is encapsulated by an instance of the APDU class, unique for each session, called the APDU object. The applet receives the `APDU` object from the JCRE as an argument of the `Appplet.process(APDU)`, invoked by the JCRE. Thus, the APDU object is used as a communication object between applet and JCRE. Command data is copied into the APDU buffer and response data is sent to the JCRE from the APDU buffer by the applet after invoking the appropiate `APDU` API methods on the current `APDU` Object (Appendix D).

The APDU object is owned by the JCRE. It is also a temporary JCRE Entry Point Object, which allows access from any applet context but prevents it from being stored in class variables, instance variables (fields) or array components. Similarly , the APDU buffer is a JCRE-owned global array.

The location of the current APDU object can be obtained from the JCRE APDU element with the *locAPDU* function. The location of the APDU buffer encapsulated by an APDU object can be extracted using the *locBuffer* function. Thus, current APDU object o and APDU buffer (array) are obtained from JCRE element J by writing $H(J.apduCmp.locAPDU)$ and $o.locBuffer$, respectively.

The APDU object is equipped with a set of functions that return information about the command/response communication session that is taken place. In particular, the *state* function returns the state of the APDU object (APDUState domain, Figure 5), which is checked and changed by the `APDU` API methods.

10.3 Receiving a Command

The applet receives the location of the APDU object as an argument to the `Applet.process(APDU)` method. The JCRE prepares the APDU object by copying the first 5 bytes of the APDU buffer with the contents of the header of the APDU command. This operation is defined by *receiveHeader*:

$$
\begin{array}{l}
\mathit{receiveHeader} \in \mathsf{APDUCmp} \rightarrow (\mathsf{CAD} \times \mathsf{Heap} \rightarrow \mathsf{Heap}) \\
\mathit{apdu}.\mathit{receiveHeader}(\mathit{cad}, H) = H' \Leftrightarrow \\
\qquad o = H(\mathit{apdu}.\mathit{locAPDU}) \wedge a = H(o.\mathit{locBuffer}) \wedge \\
\qquad (a' \in \mathsf{ArrayObject}) \wedge (a'.\mathit{length} = a.\mathit{length}) \wedge (\forall i.\ 5 \leq i < a.\mathit{length}.a'[i] = a[i]) \wedge \\
\qquad (\mathit{cad}.\mathit{message} \models^{\text{apdu}}_{\text{header}} a') \wedge o' = o[\mathit{state} \rightarrow \mathtt{IN/READY}] \\
\qquad H' = H[o.\mathit{locBuffer} \rightarrow a'][\mathit{apdu}.\mathit{locAPDU} \rightarrow o']
\end{array}
$$

where

$$
\begin{array}{l}
H \models^{\text{apdu}}_{\text{header}} apdu \Leftrightarrow apdu.CAD.message \models^{\text{apdu}}_{\text{header}} H(H(apdu.locAPDU).locBuffer) \\
msg \models^{\text{apdu}}_{\text{header}} a \Leftrightarrow \begin{array}[t]{l} (msg \in \mathsf{CommandAPDU}) \wedge \\ (a[0] = msg.CLA) \wedge (a[1] = msg.INS) \wedge (a[2] = msg.P1) \wedge (a[3] = msg.P2) \\ a[4] = \begin{cases} 0 & (msg.Lc = 0) \wedge (msg.Le = 0) \quad \text{Case 1} \\ msg.Le & (msg.Lc = 0) \wedge (msg.Le \neq 0) \quad \text{Case 2} \\ msg.Lc & (msg.Lc \neq 0) \quad \text{Case 3 or 4} \end{cases} \end{array}
\end{array}
$$

Once the header has been copied from the CAD into the APDU buffer, the user is ready to start reading the contents of the comand stored in the *ldata* field. This operation is requested by the applet by invoking the `setIncomingAndReceive()` method. This method sets the transfer direction to *Incoming* and copies data bytes of the APDU command to the APDU buffer. It takes as an argument the array index where the bytes should be copied and returns the total amount of bytes that were transferred. The specification is shown below:

$$
\frac{\begin{array}{l}
m = \texttt{javacard.framework.APDU.setIncomingAndReceive()} \\
(o = H(loc)) \wedge (a = H(o.locBuffer)) \wedge (apdu = J.apdu) \\
(o.state = \texttt{IN/READY}) \wedge (o' = o[state \to \texttt{IN/REC}]) \wedge (H' = H[loc \to o']) \\
m' = \texttt{javacard.framework.APDU.receiveBytes(short)} \\
\langle (J, K, H'), SF{::}\langle own, m, pc, V, S{::}loc{::}5\rangle\rangle, m' \overset{a}{\Rightarrow} \langle (J', K, H''), SF'\rangle
\end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S{::}loc\rangle\rangle, m \overset{a}{\Rightarrow} \langle (J', K, H''), SF'\rangle}
$$

The `setIncomingAndReceive` method will copy all the data bytes of the APDU command provided there is enough space in the APDU buffer. Any remaining bytes must be read using successive calls to the `receiveBytes` method, which is also used by the specification above to send a read request starting from position 5 of the command i.e. right after the header . The specification of `receiveBytes(short)` describes the process of reading data bytes from the CAD and copying them to the APDU buffer[3]:

$$
\frac{\begin{array}{l}
m' = \texttt{javacard.framework.APDU.receiveBytes(short)} \\
(o = H(loc)) \wedge (a = H(o.locBuffer)) \wedge (apdu = J.apdu) \\
(o.state = \texttt{IN/REC}) \wedge ((a.length - bOff) < apdu.inBlockSize) \wedge (0 \leq bOff < a.length) \\
(msg = cad.message) \wedge (free = a.length - bOff) \wedge (rem = msg.data.length - cad.ldata) \\
(((free \geq rem) \wedge (l = rem)) \vee ((free < rem) \wedge (0 < l \leq free))) \\
(cad', seq) = cad.readData(l) \wedge a' = a.getSeq(seq, bOff) \\
(J' = J.apdu[CAD \to cad']) \wedge (H' = H[o.locBuffer \to a'])
\end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S{::}loc{::}bOff\rangle\rangle, m' \overset{a}{\Rightarrow} \langle (J', K, H'), SF{::}\langle own, m, pc+1, V, S{::}l\rangle\rangle}
$$

`receiveBytes(short)` may not succeed. Failure is reported by throwing the appropiate exception:

$$
\frac{\begin{array}{l}
m' = \texttt{javacard.framework.APDU.receiveBytes()} \\
F = \langle own, m, pc, V, S{::}loc{::}bOff\rangle \\
(apdu = J.apdu) \wedge (o = H(loc) \wedge (a = H(o.locBuffer)) \\
r = \begin{cases} \texttt{APDUException.ILLEGAL\_USE} & o.state \neq \texttt{IN/REC} \\ \texttt{APDUException.BUFFER\_BOUNDS} & (a.length - bOff) < o.inBlockSize \\ \texttt{ArrayOutOfBoundsException} & (bOff < 0) \vee (bOff > a.length) \\ \texttt{APDUException.IO\_ERROR} & \texttt{I/O-error} = cad.receiveData(l) \end{cases} \\
(J', H', loc_e) = getJCException(J, H, \texttt{APDUException}, r) \\
SF' = catchException(SF{::}\langle own, m, pc, V, S\rangle, H', loc_e)
\end{array}}{\langle (J, K, H), SF{::}F\rangle, m' \overset{a}{\Rightarrow} \langle (J', K, H'), SF'\rangle}
$$

[3]The *getSeq* function extracts a sequence of bytes from a byte array. *writeSeq* copies a sequence of bytes to an array

10.4 Sending a Response

Once the applet has finished reading and processing the APDU command, it can return data to the host by preparing and sending a response APDU. The first step is to set the data transfer direction to outbound. This is done by the `setOutgoing()` method, defined as follows:

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.setOutgoing()} \\ (o = H(loc)) \wedge (o.state \in \{\texttt{IN/READY}, \texttt{IN/REC}\}) \wedge (Le = cad.message.Le) \\ (cad' \in \mathsf{CAD}) \wedge (cad'.message = \texttt{NoMsg}) \wedge (cad'.ldata = 0) \\ (o' = o[state \to \texttt{OUT/SET}]) \wedge (H' = H[loc \to o']) \wedge (J' = J.apdu[CAD \to cad']) \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S{::}loc\rangle\rangle, m' \stackrel{a}{\Rightarrow} \langle (J', K, H'), SF{::}\langle own, m, pc+1, V, S{::}Le\rangle\rangle}$$

The `setOutgoing` method sets the data transfer direction to outbound, clears any remaining data from the CAD (`NoMsg`) and returns the number of bytes expected by the host for the command APDU to which the command is responding. However, the total number of bytes sent by the applet might be different. The applet must indicate the total amount of bytes it intends to send by invoking the `setOutgoingLength(short)`:

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.setOutgoingLength(short)} \\ (o = H(loc)) \wedge (o.state = \texttt{OUT/SET}) \\ (o' = o[Le \mapsto l][state \mapsto \texttt{OUT/READY}) \wedge (H' = H[loc \to o']) \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S{::}loc{::}l\rangle\rangle, m' \stackrel{a}{\Rightarrow} \langle (J', K, H'), SF{::}\langle own, m, pc+1, V, S\rangle\rangle}$$

The `setOutgoingLength(short)` method sets the Le field of the current APDU object. The next step is to copy the response bytes from the APDU buffer to the CAD. This is done by invoking the `sendBytes(short,short)` method:

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.sendBytes(short,short)} \\ (o = H(loc)) \wedge (o.state \in \{\texttt{OUT/READY}, \texttt{OUT/SEND}\}) \wedge (a = H(o.locBuffer)) \\ seq = a.getSeq(off, l) \wedge cad' = cad.writeSeq(seq) \\ (o' = o[Le \mapsto l][state \mapsto \texttt{OUT/SEND}) \wedge (H' = H[loc \to o']) \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S{::}loc{::}off{::}l\rangle\rangle, m' \stackrel{a}{\Rightarrow} \langle (J', K, H'), SF{::}\langle own, m, pc+1, V, S\rangle\rangle}$$

where off indicates the offset where the copying of the l bytes should start.

10.5 Other APDU methods

$$\frac{m' = \texttt{javacard.framework.getBuffer()} \wedge (v = H(loc).locBuffer)}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S{::}loc\rangle\rangle, m' \stackrel{a}{\Rightarrow} \langle (J', K, H), SF{::}\langle own, m, pc+1, V, S{::}v\rangle\rangle}$$

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.getInBlockSize()}\wedge \\ (apdu = J.padu) \wedge v = \left\{ \begin{array}{ll} 1 & apdu.protocol = \texttt{T0} \\ IFSC & apdu.protocol = \texttt{T1} \end{array} \right. \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S\rangle\rangle, m' \stackrel{a}{\Rightarrow} \langle (J', K, H), SF{::}\langle own, m, pc+1, V, S{::}v\rangle\rangle}$$

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.getOutBlockSize()}\wedge \\ (apdu = J.padu) \wedge v = \left\{ \begin{array}{ll} 258 & apdu.protocol = \texttt{T0} \\ apdu.IFSD & apdu.protocol = \texttt{T1} \end{array} \right. \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S\rangle\rangle, m' \stackrel{a}{\Rightarrow} \langle (J', K, H), SF{::}\langle own, m, pc+1, V, S{::}v\rangle\rangle}$$

11 Conclusions and Future Work

We have presented the operational semantics of Java Card as a transition system between the different states of the card which respond to actions of the outside world i.e. events, and interacts with the different components pf the JCRE. This top-level semantics is the glue that puts together the Java Card components (JCRE, JCVM and API) and relates them to the outside world (host).

The specification of the JCRE components provides us with the elements neccesary to build the complete specification of the `javacard.framework` API methods. We have given the specification of the most important methods.

The specification of the Java Card Semantics comprises the most important events and actions of the card, with the exception of applet installation. Future work should address this important issue. It would also be desirable to establish notions of consistency: relations that describe the possible states of the card and the relation between its components.

A Exceptions

- Constructors:

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.}e\texttt{.<init>(short)} \\ e \in \mathsf{JCCheckedExceptions} \\ o = H(loc).[reason \mapsto r] \wedge H' = H[loc \mapsto o] \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S{::}loc{::}r\rangle\rangle, m' \overset{a}{\Rightarrow} \langle (J, K, H), SF{::}\langle own, m, pc+1, V, S\rangle\rangle}$$

- `public short getReason()`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.}e\texttt{.getReason()} \\ e \in \mathsf{JCCheckedExceptions} \wedge r = H(loc).reason \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S{::}loc\rangle\rangle, m' \overset{a}{\Rightarrow} \langle (J, K, H), SF{::}\langle own, m, pc+1, V, S{::}r\rangle\rangle}$$

- `public void setReason(short)`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.}e\texttt{.setReason()} \\ e \in \mathsf{JCCheckedExceptions} \wedge o = H(loc).[reason \mapsto r] \wedge H' = H[loc \mapsto o] \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S{::}loc{::}r\rangle\rangle, m' \overset{a}{\Rightarrow} \langle (J, K, H'), SF{::}\langle own, m, pc+1, V, S\rangle\rangle}$$

- `public static void throwIt(short)`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.}e\texttt{.throwIt(short)} \\ e \in \mathsf{JCCheckedExceptions} \wedge F = \langle own, m, pc, V, S{::}r\rangle \\ E = J.exceptionCmp \wedge (E', H', loc_e) = E.getJCException(H, e, r) \\ J' = J[exceptionCmp \mapsto E'] \wedge SF' = catchException(SF{::}F, H', loc_e) \end{array}}{\langle (J, K, H), SF{::}F\rangle, m' \overset{a}{\Rightarrow}_{\mathrm{st}} \langle (J', K, H'), SF'\rangle}$$

B The `AID` class

- `AID(byte[] bArray, short offset, byte length)`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.AID.<init>(byte[], short, byte)} \\ (loc_a \neq \texttt{null}) \wedge (0 \leq \mathit{offset} \leq H(loc_a).\mathit{length} - l) \\ (5 \leq l \leq 16) \wedge (\mathit{aid}.\mathit{length} = l) \wedge \forall i, 0 \leq i < l, \mathit{aid}[i] = H(loc_a).\mathit{values}[\mathit{offset} + i] \\ o = H(loc)[\mathit{theAID} \mapsto \mathit{aid}] \wedge H' = H[loc \mapsto o] \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S{::}loc{::}loc_a{::}\mathit{offset}{::}l\rangle\rangle, m' \stackrel{a}{\Rightarrow} \langle (J, K, H), SF{::}\langle own, m, pc+1, V, S\rangle\rangle}$$

Runtime Exception:

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.AID.<init>(byte[], short, byte)} \\ F = \langle own, m, pc, V, S{::}loc{::}loc_a{::}\mathit{offset}{::}l\rangle \\ e = \begin{cases} \texttt{NullPointerException} & (loc_a = \texttt{null}) \\ \texttt{ArrayIndexOutOfBoundsException} & (\mathit{offset} < 0) \vee (\mathit{offset} > H(loc_a).\mathit{length} - l) \\ \texttt{SystemException} & (l < 5) \vee (l > 16) \end{cases} \\ E = J.\mathit{exceptionCmp} \\ (E', H', loc_e) = \begin{cases} E.\mathit{getJCException}(H, e, \texttt{ILLEGAL\_VALUE}) & e = \texttt{SystemException} \\ E.\mathit{getException}(H, e) & \text{otherwise} \end{cases} \\ J' = J[\mathit{exceptionCmp} \mapsto E'] \wedge SF' = \mathit{catchException}(SF{::}\langle own, m, pc, V, S\rangle, H', loc_e) \end{array}}{\langle (J, K, H), SF{::}F\rangle, m' \stackrel{a}{\Rightarrow} \langle (J', K, H'), SF'\rangle}$$

- `boolean equals(byte[] bArray, short offset, byte length)`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.AID.equals(byte[], short, byte)} \\ F = \langle own, m, pc, V, S{::}loc{::}loc_a{::}\mathit{offset}{::}l\rangle \\ (loc_a \neq \texttt{null}) \Rightarrow (0 \leq \mathit{offset} \leq H(loc_a).\mathit{length} - l) \\ v = \begin{cases} \texttt{true} & (loc_a \neq \texttt{null}) \wedge (\mathit{aid} = H(loc).\mathit{theAID}) \\ & (l = \mathit{aid}.\mathit{length}) \wedge \forall i, 0 \leq i < l, \mathit{aid}[i] = H(loc_a).\mathit{values}[\mathit{offset} + i] \\ \texttt{false} & \text{otherwise} \end{cases} \end{array}}{\langle (J, K, H), SF{::}F\rangle, m' \stackrel{a}{\Rightarrow} \langle (J, K, H), SF{::}\langle own, m, pc+1, V, S{::}v\rangle\rangle}$$

Runtime Exception:

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.AID.equals(byte[], short, byte)} \\ F = \langle own, m, pc, V, S{::}loc{::}loc_a{::}\mathit{offset}{::}l\rangle \\ (loc_a \neq \texttt{null}) \wedge ((\mathit{offset} \leq 0) \vee (\mathit{offset} > H(loc_a).\mathit{length}) - l) \\ E = J.\mathit{exceptionCmp} \wedge J' = J[\mathit{exceptionCmp} \mapsto E'] \\ (E', H', loc_e) = E.\mathit{getException}(H, \texttt{ArrayIndexOutOfBoundsException}) \\ SF' = \mathit{catchException}(SF{::}\langle own, m, pc, V, S\rangle, H', loc_e) \end{array}}{\langle (J, K, H), SF{::}F\rangle, m' \stackrel{a}{\Rightarrow} \langle (J', K, H'), SF'\rangle}$$

- `boolean equals(Object anObject)`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.AID.equals(Object)} \\ v = \begin{cases} \texttt{true} & (loc' \neq \texttt{null}) \wedge H(loc).\mathit{theAID} = H(loc').\mathit{theAID} \\ \texttt{false} & \text{otherwise} \end{cases} \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S{::}loc{::}loc'\rangle\rangle, m' \stackrel{a}{\Rightarrow} \langle (J, K, H), SF{::}\langle own, m, pc+1, V, S{::}v\rangle\rangle}$$

- `byte getBytes(byte[] dest, short offset)`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.AID.getBytes(byte[], short)} \\ (aid = H(loc).theAID) \wedge (l = aid.length) \\ (loc_a \neq \texttt{null}) \wedge (0 \leq offset \leq H(loc_a).length - l) \\ a = H(loc_a).\mathrm{values}[offset + i \mapsto aid[i]],\ 0 \leq i < l \\ H' = H[loc_a \mapsto a] \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S{::}loc{::}loc_a{::}offset\rangle\rangle, m' \overset{a}{\Rightarrow} \langle (J, K, H), SF{::}\langle own, m, pc+1, V, S{::}l\rangle\rangle}$$

Runtime Exception:

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.AID.getBytes(byte[], short)} \\ F = \langle own, m, pc, V, S{::}loc{::}loc_a{::}offset\rangle \\ (aid = H(loc).theAID) \wedge (l = aid.length) \\ e = \begin{cases} \texttt{NullPointerException} & (loc_a = \texttt{null}) \\ \texttt{ArrayIndexOutOfBoundsException} & (offset < 0) \vee (offset > H(loc_a).length - l) \end{cases} \\ E = J.exceptionCmp \wedge J' = J[exceptionCmp \mapsto E'] \\ (E', H', loc_e) = E.getException(H, e) \\ SF' = catchException(SF{::}\langle own, m, pc, V, S\rangle, H', loc_e) \end{array}}{\langle (J, K, H), SF{::}F\rangle, m' \overset{a}{\Rightarrow} \langle (J', K, H'), SF'\rangle}$$

- `boolean partialEquals(byte[] bArray, short offset, byte length)`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.AID.partialEquals(byte[], short, byte)} \\ F = \langle own, m, pc, V, S{::}loc{::}loc_a{::}offset{::}l\rangle \\ (loc_a \neq \texttt{null}) \Rightarrow (0 \leq offset \leq H(loc_a).length - l) \\ v = \begin{cases} \texttt{true} & (loc_a \neq \texttt{null}) \wedge (aid = H(loc).theAID) \\ & (l \leq aid.length) \wedge \forall i, 0 \leq i < l, aid[i] = H(loc_a).\mathrm{values}[offset + i] \\ \texttt{false} & \text{otherwise} \end{cases} \end{array}}{\langle (J, K, H), SF{::}F\rangle, m' \overset{a}{\Rightarrow} \langle (J, K, H), SF{::}\langle own, m, pc+1, V, S{::}v\rangle\rangle}$$

Runtime Exception:

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.AID.partialEquals(byte[], short, byte)} \\ F = \langle own, m, pc, V, S{::}loc{::}loc_a{::}offset{::}l\rangle \\ (loc_a \neq \texttt{null}) \wedge ((offset < 0) \vee (offset > H(loc_a).length - l)) \\ E = J.exceptionCmp \wedge J' = J[exceptionCmp \mapsto E'] \\ (E', H', loc_e) = E.getException(H, \texttt{ArrayIndexOutOfBoundsException}) \\ SF' = catchException(SF{::}\langle own, m, pc, V, S\rangle, H', loc_e) \end{array}}{\langle (J, K, H), SF{::}F\rangle, m' \overset{a}{\Rightarrow} \langle (J', K, H'), SF'\rangle}$$

- `boolean RIDEquals(AID otherAID)`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.AID.RIDEquals(AID)} \\ aid = H(loc).theAID \\ v = \begin{cases} \texttt{true} & (loc' \neq \texttt{null}) \wedge (aid' = H(loc').theAID) \\ & RIDEquals(aid, aid') \\ \texttt{false} & \text{otherwise} \end{cases} \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S{::}loc{::}loc'\rangle\rangle, m' \overset{a}{\Rightarrow} \langle (J, K, H), SF{::}\langle own, m, pc+1, V, S{::}v\rangle\rangle}$$

```
public final class JCSystem {
  /* state information */
  public static short getVersion();
  public static AID getAID();
  public static Shareable getAppletShareableInterfaceObject(AID, byte);
  public static AID getPreviousContextAID();
  public static AID lookupAID(byte[], short, byte);
  /* Transactions */
  public static native void abortTransaction();
  public static native void beginTransaction();
  public static native void commitTransaction();
  public static native short getMaxCommitCapacity();
  public static native byte getTransactionDepth();
  public static native short getUnusedCommitCapacity();
  /* Transient Objects */
  public static native byte isTransient(Object);
  public static native boolean[] makeTransientBooleanArray(short, byte);
  public static native byte[] makeTransientByteArray(short, byte);
  public static native Object[] makeTransientObjectArray(short, byte);
  public static native short[] makeTransientShortArray(short, byte);
}
```

Figure 6: The `JCSystem` Class

C The `JCSystem` class

The `JCSystem` class includes a collection of methods to:

- Control applet execution.
- Resource management.
- Atomic transaction management.
- Inter-applet object sharing.
- Control the persistence and transience of objects.

The methods[4] declared in the `JCSystem` are listed in Figure 6.

C.1 Applet and AID Information

- `public static AID getAID()`

$$\frac{m' = \texttt{javacard.framework.JCSystem.getAID()} \qquad (J', H', loc) = J.\mathit{getAID}(H, \mathit{own}.\mathit{aid})}{\langle (J,K,H), SF{::}\langle \mathit{own}, m, pc, V, S\rangle\rangle, m' \overset{a}{\Rightarrow}_{\mathrm{st}} \langle (J',K,H'), SF{::}\langle \mathit{own}, m, pc+1, V, S{::}\mathit{loc}\rangle\rangle}$$

[4] The rules listed in the remaining of this document have not been updated with the new JCRE specification based on components.

- `public static AID getPreviousContextAID()`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.getPreviousContextAID()} \\ own' = getPreviousContext(SF::\langle own, m, pc, V, S\rangle, own) \\ (J', H', loc) = \begin{cases} (J, H, \texttt{null}) & own' = \texttt{JCRE} \\ J.getAID(H, own.aid) & \text{otherwise} \end{cases} \end{array}}{\langle (J, K, H), SF::\langle own, m, pc, V, S\rangle\rangle, m' \stackrel{a}{\Rightarrow}_{\mathrm{st}} \langle (J', K, H'), SF::\langle own, m, pc+1, V, S::loc\rangle\rangle}$$

- `public static AID lookupAID(byte[], short, byte)`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.lookupAID(byte[], short, byte)} \\ (loc \neq \texttt{null}) \wedge (0 \leq offset \leq H(loc).length - l) \\ aid = H(loc).values[offset, offset + l] \\ (J', H', loc) = \begin{cases} (J, H, \texttt{null}) & own' = \texttt{JCRE} \\ J.getAID(H, aid) & \text{otherwise} \end{cases} \end{array}}{\langle (J, K, H), SF::\langle own, m, pc, V, S::loc::offset::l\rangle\rangle, m' \stackrel{a}{\Rightarrow}_{\mathrm{st}} \langle (J', K, H'), SF::\langle own, m, pc+1, V, S::loc'\rangle\rangle}$$

Runtime Exception:

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.lookupAID(byte[], short, byte)} \\ F = \langle own, m, pc, V, S::loc::offset::l\rangle \\ e = \begin{cases} \texttt{NullPointerException} & (loc = \texttt{null}) \\ \texttt{ArrayIndexOutOfBoundsException} & (offset < 0) \wedge (offset > H(loc).length - l) \end{cases} \\ (J', H', loc_e) = getException(J, H, e) \\ SF' = catchException(SF::\langle own, m, pc, V, S\rangle, H', loc_e) \end{array}}{\langle (J, K, H), SF::F\rangle, m' \stackrel{a}{\Rightarrow}_{\mathrm{st}} \langle (J', K, H'), SF'\rangle}$$

C.2 Transactions

- `public static native void abortTransaction()`

  Aborts the atomic transaction. The contents of the commit buffer is discarded. The `JCRE` ensures that any variable of reference type which references an object instantiated from within this aborted transaction is equivalent to a `null` reference. The `JCRE` may not recover the heap space used by the new objects instances created from within the aborted transaction. Furthermore, the `JCRE` may lock up the card session to force tear/reset processing in order to ensure security of the card and to avoid heap space loss.

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.abortTransaction()} \\ J.transactionDepth \neq 0 \\ H' = restoreHeap(J.cb, H) \wedge (V', S') = restoreRefs(J, H, V, S) \\ J' = J[cb \mapsto \perp_{cb}][transactionDepth \mapsto 0] \end{array}}{\langle (J, K, H), SF::\langle own, m, pc, V, S\rangle\rangle, m' \stackrel{a}{\Rightarrow}_{\mathrm{st}} \langle (J', K, H'), SF::\langle own, m, pc+1, V, S\rangle\rangle}$$

Runtime Exception:

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.abortTransaction()} \\ J.transactionDepth = 0 \\ (J', H', loc_e) = getException(J, H, \texttt{TransactionException}) \\ H'(loc).reason = \texttt{NOT\_IN\_PROGRESS} \\ SF' = catchException(SF{::}\langle own, m, pc, V, S\rangle, H', loc_e) \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S\rangle\rangle, m' \overset{a}{\Rightarrow}_{\mathrm{st}} \langle (J', K, H'), SF'\rangle}$$

- `public static native void beginTransaction()`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.beginTransaction()} \\ J.transactionDepth = 0 \land J' = J[transactionDepth \mapsto 1] \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S\rangle\rangle, m' \overset{a}{\Rightarrow}_{\mathrm{st}} \langle (J', K, H), SF{::}\langle own, m, pc+1, V, S\rangle\rangle}$$

Runtime Exception:

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.beginTransaction()} \\ J.transactionDepth \neq 0 \\ (J', H', loc_e) = getException(J, H, \texttt{TransactionException}) \\ H'(loc).reason = \texttt{IN\_PROGRESS} \\ SF' = catchException(SF{::}\langle own, m, pc, V, S\rangle, H', loc_e) \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S\rangle\rangle, m' \overset{a}{\Rightarrow}_{\mathrm{st}} \langle (J', K, H'), SF'\rangle}$$

- `public static native void commitTransaction()`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.commitTransaction()} \\ (d = J.transactionDepth) \land (d \neq 0) \land (J' = J[transactionDepth \mapsto (d-1)]) \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S\rangle\rangle, m' \overset{a}{\Rightarrow}_{\mathrm{st}} \langle (J', K, H), SF{::}\langle own, m, pc+1, V, S\rangle\rangle}$$

Runtime Exception:

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.commitTransaction()} \\ J.transactionDepth = 0 \\ (J', H', loc_e) = getException(J, H, \texttt{TransactionException}) \\ H'(loc).reason = \texttt{NOT\_IN\_PROGRESS} \\ SF' = catchException(SF{::}\langle own, m, pc, V, S\rangle, H', loc_e) \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S\rangle\rangle, m' \overset{a}{\Rightarrow}_{\mathrm{st}} \langle (J', K, H'), SF'\rangle}$$

- `public static native short getMaxCommitCapacity()`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.getMaxCommitCapacity} \\ v = J.maxCommitCapacity \end{array}}{\langle (J, K, H), \langle own, m, pc, V, S\rangle\rangle, m' \overset{a}{\Rightarrow}_{\mathrm{st}} \langle (J, K, H), SF{::}\langle own, m, pc+1, V, S{::}v\rangle\rangle}$$

- `public static native byte getTransactionDepth()`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.getTransactionDepth()} \\ v = J.transactionDepth \end{array}}{\langle (J, K, H), \langle own, m, pc, V, S\rangle\rangle, m' \overset{a}{\Rightarrow}_{\mathrm{st}} \langle (J, K, H), SF{::}\langle own, m, pc+1, V, S{::}v\rangle\rangle}$$

- `public static native short getUnusedCommitCapacity()`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.getUnusedCommitCapacity} \\ v = J.maxCommitCapacity - usedCommitCapacity(J.cb) \end{array}}{\langle (J, K, H), \langle own, m, pc, V, S\rangle\rangle, m' \overset{a}{\Rightarrow}_{\text{st}} \langle (J, K, H), SF{::}\langle own, m, pc+1, V, S{::}v\rangle\rangle}$$

C.3 Transitivity

- `public static native byte isTransient(Object)`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.isTransient(Object)} \\ v = \begin{cases} H(loc).transient & loc \neq \texttt{null} \wedge H(loc) \in \mathsf{ArrayObject} \\ \texttt{NOT\_TRANSIENT} & \text{otherwise} \end{cases} \end{array}}{\langle G, SF{::}\langle own, m, pc, V, S{::}loc\rangle\rangle, m' \overset{a}{\Rightarrow} \langle G, SF{::}\langle own, m, pc+1, V, S{::}v\rangle\rangle}$$

- `public static native boolean[] makeTransientBooleanArray(short, byte)`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.makeTransientBooleanArray(short, byte)} \\ e \in \{\texttt{CLEAR\_ON\_RESET}, \texttt{CLEAR\_ON\_DESELECT}\} \\ (e = \texttt{CLEAR\_ON\_DESELECT}) \Rightarrow (own.context = getAppletContext(SF{::}\langle own, m, pc, V, S{::}l{::}e\rangle)) \\ (H', loc) = newArray(own, \texttt{bool}, l, \texttt{false}, e, H) \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S{::}l{::}e\rangle\rangle, m' \overset{a}{\Rightarrow}_{\text{st}} \langle (J, K, H'), SF{::}\langle own, m, pc+1, V, S{::}loc\rangle\rangle}$$

Runtime Exception:

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.makeTransientBooleanArray(short, byte)} \\ r = \begin{cases} \texttt{ILLEGAL\_VALUE} & e \notin \{\texttt{CLEAR\_ON\_RESET}, \texttt{CLEAR\_ON\_DESELECT}\} \\ \texttt{ILLEGAL\_TRANSIENT} & (e = \texttt{CLEAR\_ON\_DESELECT}) \wedge \\ & (own.context \neq getAppletContext(SF{::}\langle own, m, pc, V, S{::}l{::}e\rangle)) \\ \texttt{NO\_TRANSIENT\_SPACE} & err = newArray(own, \texttt{bool}, l, \texttt{false}, e, H) \end{cases} \\ (J', H', loc_e) = getJCException(J, H, \texttt{SystemException}, r) \\ SF' = catchException(SF{::}\langle own, m, pc, V, S\rangle, H', loc_e) \end{array}}{\langle (J, K, H), SF{::}\langle own, m, pc, V, S\rangle\rangle, m' \overset{a}{\Rightarrow}_{\text{st}} \langle (J', K, H'), SF'\rangle}$$

References

[1] Renaud Marlet. Personal communication.

[2] Igor Siveroni. Operational semantics of the Java Card virtual machine. *Journal of Logic and Algebraic Programming, special issue on Smart Cards*, (To appear), 2003. SECSAFE-ICSTM-011.

[3] Igor Siveroni and Chris Hankin. A proposal of the JCVMLe operational semantics. Technical Report SECSAFE-ICSTM-001, Imperial College London, 2002. http://www.doc.ic.ac.uk/˜siveroni/secsafe.

[4] Sun Microsystems, Inc., Palo Alto, CA, USA. *Java Card 2.1.1 Platform API Specification*, May 2000.

[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 2000.

[6] 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.