Formalisation of the Java Card Runtime Environment and API

Author:	Igor Siveroni
Date:	November, 2002
Number:	SECSAFE-ICSTM-006
Version:	1.0
Classification:	Public
Status:	Work in progress

1 Introduction

We present a formal specification of the operational semantics of the Java Card API and JCRE. The Java Card platform is defined by three specifications: The *Java Card Virtual Machine* (JCVM) *Specification* [5], the *Java Card Application Programming Interface* (API) [4], and the *Java Card Runtime Environment* (JCRE) *Specification* [3]. 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.

The Java Card API consists of three packages:

- `java.lang` package: Provides classes that are fundamental to the design of the Java Card technology subset of the Java programming language.
- `javacard.framework` package: Provides framework of classes and interfaces for the core functionality of a Java Card applet.

- `javacard.security` package: Provides the classes and interfaces for the Java Card security framework.
- `javacardx.crypto` package: Extension package containing security classes and interfaces for export-controlled functionality.

  This document comprises the specification of the `java.lang` and `javacard.framework` packages. We have covered most of the important JCRE and API features: exceptions (section 4), applets and AIDs (section 5), transactions (section 6) and transient objects (section 7). We are still working with the definition of the JCRE features concerned with the manipulation of APDU messages (`APDU` class) sent to and received from off-card applications, and the integration of all these features in a top-level semantics of the JCRE. The complete specification will appear in the next version of this document.

2 Semantics

API methods are specified as special cases of virtual and static method invocations, based on the semantics of the JCVM given in Secsafe-ICSTM-001 [2]. 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 JCRE element is specified as a domain equipped with a series of functions that determine a particular feature of the Java Card platform. For example, the value associated with the depth of a transaction can be obtained by applying the function

$$\mathit{transactionDepth} \in \mathsf{JCRE} \to \mathsf{ByteValue}$$

to the JCRE element i.e. *jcre.transactionDepth*.

The execution of API methods is defined by the functions:

$$\begin{array}{ll} -,-\ \overset{a}{\Rightarrow}\ - & \in \mathit{Config} \times \mathsf{Method} \to \mathit{Config} \\ -,-\ \overset{a}{\Rightarrow}_{\text{st}}\ - & \in \mathit{Config} \times \mathsf{Method} \to \mathit{Config} \end{array}$$

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 method invocations from the JCVM looks like:

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

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

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

```
• Object
    ∘ Throwable
        • Exception
                                  | ArithmeticException
                                  | ArrayStoreException
                                  | ClassCastException
                                  | IndexOutOfBoundsException
            ∘ RuntimeException    |   ∘ArrayIndexOutOfBoundsException
                                  | NegativeArraySizeException
                                  | NullPointerException
                                  | SecurityException
```

Figure 1: `java.lang` package class hierarchy

3 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 1. It provides the definition of the `Object` class, root of the Java Card class hierarchy, and `Throwable` class, superclass of all exceptions and errors.

3.1 The `Object` class

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

- `public Object()`

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

- `public boolean equals(Object)`

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

3.2 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[1]:

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

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

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:

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

The semantics of throwing and catching an exception are defined in [2].

4 Exceptions

4.1 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 JCExceptions - are :

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

The 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:

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

4.2 Exceptions and the JCRE

Being memory usage an important concern, the creation of unique runtime exception objects per class is enforced. The JCRE keeps a reference to each instance of the Java runtime exception objects in the *jcre.eInst* element defined by:

We cover both cases by introducing the *eInst* element, a function that maps (Java runtime and Java Card) an exception class to the location of the respective (unique) exception object. We have:

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

A mapping *eInst* is well-formed with respect to heap H - we write $H \models eInst$ -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:

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

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.

$$getException(jcre, H, cl) = \begin{cases} (jcre, H, jcre.eInst(cl)) & \text{if } cl \in dom(jcre.eInst) \\ (jcre.eInst[cl \mapsto loc], H', loc) & \text{otherwise} \\ \quad \text{where } (H', loc) = newObject(\texttt{JCRE}, cl, tEP, H) & \end{cases}$$

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 $H \models eInst$ and *getException*, instances of the Java Card exception classes defined in 4.1 are associated with a *reason* code. Given reason code r, $getJCException(jcre, H, e, r)$ returns the unique instance of class e kept by the `JCRE` with the *reason* state variable set to r.

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

The `throwit` method is defined by:

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

A detailed description of the to JCExceptions class methods can be found in appendix A.

5 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.

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

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.

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

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 8) during the installation process (section 9).

5.1 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 8). We model the process of applet registration by defining a special structure to keep track of all registered applets, the applet table:

$$\begin{array}{l} appTable : \mathsf{JCRE} \mapsto \mathsf{AppletTable} \\ applets \in \mathsf{AppletTable} = \mathsf{AID} \xrightarrow{1-1} \mathsf{Location} \end{array}$$

The applet table is a one-to-one mapping from elements of the AID domain to locations and can be obtained from the JCRE using the *appTable* JCRE interface function. The object locations registered in an applet 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 process of applet installation, registration, selection and execution is explained in detail in section 9.

5.2 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:

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

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.

5.3 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[3].

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:

$$aids \in \mathsf{AIDTable} = \mathsf{AID} \mapsto \mathsf{Location}$$

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:

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

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

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

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:

$$getAID : (\mathsf{JCRE} \times \mathsf{Heap} \times \mathsf{AID}) \rightarrow (\mathsf{JCRE} \times H \times \mathsf{Location})$$

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

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 $dom(aidTable) = dom(appTable)$. In that case, an AID object will be created by the JCRE everytime a new applet is registered.

5.4 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.

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 call stack. This search is defined by the *getPreviousContext* function:

$$getPreviousContext : \mathsf{Frame}^* \times \mathsf{Owner} \rightarrow \mathsf{Owner}$$
$$getPreviousContext(\epsilon, own') = JCRE$$
$$getPreviousContext(SF{::}\langle own, m, pc, V, S\rangle, own') = \begin{cases} own \text{ if } own.aid \neq own.'aid \\ getPreviousContext(SF, own) \text{ otherwise} \end{cases}$$

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.

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

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

6 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 with methods that guarantee atomicity for block updates of array elements (section **??**).

6.1 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 are:

```
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();
```

An applet designates the beginning and (normal) termination of a transaction with calls to the `begin-Transaction` 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 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 *transaction-Depth* is modified by the calls to `beginTransaction`, where it is incremented by one, and by calls to

$$\begin{array}{rcl}
\mathit{transactionDepth} & \in & \mathsf{JCRE} \rightarrow \mathsf{ByteValue} \\
\mathit{maxCommitCapacity} & \in & \mathsf{JCRE} \rightarrow \mathsf{ShortValue} \\
\\
\mathit{fieldStoreTransaction} & \in & (\mathsf{JCRE} \times H \times \mathsf{Location} \times \mathsf{FieldID} \times \mathsf{Value}) \rightarrow \mathsf{JCRE} \\
\mathit{arrayStoreTransaction} & \in & (\mathsf{JCRE} \times H \times \mathit{loc} \times \mathsf{ShortValue} \times \mathsf{Value}) \rightarrow \mathsf{JCRE} \\
\mathit{restoreNewObjects} & \in & \mathsf{JCRE} \times \mathsf{Heap} \rightarrow (\mathsf{Heap} \times \mathsf{LocalVar} \times \mathsf{OperandStack}) \\
\mathit{restoreUpdates} & \in & (\mathsf{JCRE} \times \mathsf{Heap} \times \mathsf{LocalVar} \times \mathsf{OperandStack}) \rightarrow (\mathsf{LocalVar} \times \mathsf{OperandStack}) \\
\\
\mathit{commitBuffer} & \in & \mathsf{JCRE} \rightarrow \mathsf{CommitBuffer} \\
\mathit{cb} \in \mathsf{CommitBuffer} & = & \mathsf{ElementLocation} \rightarrow \mathsf{Value}
\end{array}$$

Table 1: Elements of the Specification of Transactions

`commitTransaction` and `abortTransaction`, where it is decremented by one. For example, the result of applying the `beginTransaction` method, as described by the rule shon above,

$$\frac{m' = \texttt{javacard.framework.JCSystem.beginTransaction()} \qquad \mathit{jcre}.\mathit{transactionDepth} = 0 \wedge \mathit{jcre}' = \mathit{jcre}[\mathit{transactionDepth} \mapsto 1]}{\langle\!\langle (\mathit{jcre}, K, H), \mathit{SF}{::}\langle \mathit{own}, m, \mathit{pc}, V, S\rangle\!\rangle, m' \overset{a}{\Rightarrow}_{\mathrm{st}} \langle\!\langle (\mathit{jcre}', K, H), \mathit{SF}{::}\langle \mathit{own}, m, \mathit{pc}+1, V, S\rangle\!\rangle}$$

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:

6.1.1 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.
- If a transaction aborts, a read operation to 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:

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

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,

- $fieldStoreTransaction \in (\mathsf{JCRE} \times H \times \mathsf{Location} \times \mathsf{FieldID} \times \mathsf{Value}) \to \mathsf{JCRE}$:
  $fieldStoreTransaction(jcre, H, loc, id, v) = jcre[commitBuffer \mapsto cb'],$
  $$cb' = \begin{cases} updateBuffer(jcre.commitBuffer, (loc, id), v) & \text{if } (jcre.transactionDepth \neq 0) \\ jcre.cb & \text{otherwise} \end{cases},$$

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:

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

6.1.2 Abort and Transaction Failure

The atomic transaction model must ensure that all updates performed to 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 above - 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.

$$\begin{aligned}
&\mathit{restoreUpdates}(\mathit{jcre}, H) = (H', K') \text{ where } \mathit{old} = \mathit{jcre.commitBuffer.oldValues} \wedge \\
&\quad H' = H[\mathit{eloc}_1 \mapsto \mathit{old}(\mathit{eloc}_1)]\ldots[\mathit{eloc}_n \mapsto \mathit{old}(\mathit{eloc}_n)] \wedge \{\mathit{eloc}_1, \ldots, \mathit{eloc}_n\} = \mathit{dom}(\mathit{old}) \backslash \mathsf{StaticElement} \\
&\quad K' = K[\mathit{eloc}'_1 \mapsto \mathit{old}(\mathit{eloc}'_1)]\ldots[\mathit{eloc}'_n \mapsto \mathit{old}(\mathit{eloc}'_m)] \wedge \{\mathit{eloc}_1, \ldots, \mathit{eloc}_n\} = \mathit{dom}(\mathit{old}) \cap \mathsf{StaticElement}
\end{aligned}$$

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.

$$\begin{aligned}
&\mathit{restoreNewObjects}(\mathit{jcre}, H, V, S) = (H', V', S') \text{ where} \\
&\quad \mathit{locs} = \mathit{jcre.commitBuffer.newLocations} \\
&\quad V.\mathit{update}(\mathit{values}) \wedge \mathit{values} = \{(i, \texttt{null}) \mid (0 \leq i < V.\mathit{length}) \wedge V[i] \in \mathit{locs}\} \\
&\quad S.\mathit{update}(\mathit{values}) \wedge \mathit{values} = \{(i, \texttt{null}) \mid (0 \leq i < S.\mathit{length}) \wedge S[i] \in \mathit{locs}\} \\
&\quad \mathit{dom}(H') = \mathit{dom}(H) \backslash \mathit{locs} \;\wedge\; \forall \mathit{loc} \in \mathit{dom}(H').\ H'(\mathit{loc}) = \mathit{clearObject}(H(\mathit{loc}), \mathit{locs})
\end{aligned}$$

where

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

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.

7 Transient Objects

Applets sometimes require objects that contain temporary (transient) 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 [2]. This information is provided when a new transient object is allocated, as in the rule below that describes the creation of transient boolean arrays:

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

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.

7.1 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 T:

$$\mathit{clearTransient}(H,T) = H' \Leftrightarrow \begin{cases} \mathit{dom}(H) = \mathit{dom}(H') \wedge \\ \forall \mathit{loc} \in \mathit{dom}(H), : \\ \quad H'(\mathit{loc}) = \begin{cases} \mathit{clearArray}(H(\mathit{loc})) & H(\mathit{loc}) \in \mathsf{ArrayObject} \\ & H(\mathit{loc}).\mathit{transient} = T \\ H(\mathit{loc}) & \text{otherwise} \end{cases} \end{cases}$$

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

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

8 The APDU

Communication between the card and the off-card applications is performed using the Application Protocol Data Unit (APDU) format, as defined in ISO specification 7816-4. The functionality related to the use of APDUs is defined by the `APDU` class.

We are currently working on the specification of the APDU class..

9 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 2). 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.

We are currently working on the specification of the top level semantics of the JCRE and its interaction with the execution of the methods of the the Applet class.

10 Future Work

We have presented the operational semantics of JCVM and parts of the Java Card API. Both elements, in particular the API, have close interaction with the Java Card runtime environment. We are currently working on finishing the specification of the `APDU` and `Applet` classes.

```
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 2: The `Applet` class

The next step will be to define semantics of the JCRE as a transition system between the different states of a card which responds to actions of the outside world and the execution of applets in the JCVM. The JCRE, effectively the operating system of a Java Card, will be depicted as the glue that puts together its components (JCVM and API) and relates them to the outside world (host).

A Exceptions

- Constructors:

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

- `public short getReason()`

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

- `public void setReason(short)`

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

- `public static void throwIt(short)`

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

B The `AID` class

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

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

Runtime Exception:

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

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

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

Runtime Exception:

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

- `boolean equals(Object anObject)`

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

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

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

Runtime Exception:

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

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

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

Runtime Exception:

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

- `boolean RIDEquals(AID otherAID)`

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

```
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 3: 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 declared in the `JCSystem` are listed in Figure 3

C.1 Applet and AID Information

- `public static AID getAID()`

$$\frac{\begin{array}{l} m' = \texttt{javacard.framework.JCSystem.getAID()} \\ (jcre', H', loc) = jcre.getAID(H, own.aid) \end{array}}{\langle (jcre, K, H), SF{::}\langle own, m, pc, V, S\rangle\rangle, m' \overset{a}{\Rightarrow}_{\text{st}} \langle (jcre', K, H'), SF{::}\langle own, m, pc+1, V, S{::}loc\rangle\rangle}$$

- `public static AID getPreviousContextAID()`

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

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

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

Runtime Exception:

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

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.

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

Runtime Exception:

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

- `public static native void beginTransaction()`

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

Runtime Exception:

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

- `public static native void commitTransaction()`

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

Runtime Exception:

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

- `public static native short getMaxCommitCapacity()`

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

- `public static native byte getTransactionDepth()`

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

- `public static native short getUnusedCommitCapacity()`

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

C.3 Transitivity

- `public static native byte isTransient(Object)`

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

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

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

Runtime Exception:

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

D The `Util` class: Work in progress

The list of methods declared in the `Util` class are listed in Figure 4.

E The `APDU` class: Work in progress

F Interfaces

```
public abstract interface ISO7816
{
}
```

```
public class Util
{
  public static final native byte arrayCompare(byte[], short, byte[], short, short);
  public static final native short arrayCopy(byte[], short, byte[], short, short);
  public static final native short arrayCopyNonAtomic(byte[], short, byte[], short, short);
  public static final native short arrayFillNonAtomic(byte[], short, short, byte);
  public static final short getShort(byte[], short);
  public static final short makeShort(byte, byte);
  public static final native short setShort(byte[], short, short);
}
```

Figure 4: The `Util` class

```
public abstract interface PIN
{
  public abstract boolean check(byte[], short, byte);
  public abstract byte getTriesRemaining();
  public abstract boolean isValidated();
  public abstract void reset();
}

public abstract interface Shareable
{
}
```

References

[1] Renaud Marlet. Personal communication.

[2] 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.

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

[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 Virtual Machine Specification*, revision 1.0 edition, May 18 2000.