Towards a full formal specification of the JavaCard API

Hans Meijer and Erik Poll

Computing Science Institute, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
{hans,erikpoll}@cs.kun.nl

Abstract. This paper reports on ongoing work to develop a formal specification of the JavaCard API using the specification language JML. It discusses the specification of the `JCSystem` class, which deals with the JavaCard firewall, (atomic) transactions and transient objects. The `JCSystem` class seems to be the hardest class in the API to specify, and it is closely connected with some of the peculiarities of JavaCard as opposed to Java.

1 Introduction

There has been a lot of work on formalisations of the Java(Card) platform. (For a comprehensive overview see [4].) However, most of the work has concentrated on the Java(Card) Virtual Machine, and there has only been very little work on formalisations of the other component of the JavaCard platform, the JavaCard API. This paper reports on an ongoing effort to develop a formal specification of the JavaCard API using the specification language JML. Our ultimate goal is to use a formal specification of the API to verify API implementations and to use it as a basis for the verification of JavaCard applets. But of course a formal specification of the API is of wider interest, notably to improve and clarify existing informal documentation. For verification we want to use the LOOP tool developed in Nijmegen [1], which gives a formal semantics to Java programs and acts as a front-end to the theorem prover PVS. The first —very modest— steps to verify JML-annotated JavaCard code using the LOOP tool and the theorem prover PVS are reported in [2].

Earlier work on 'lightweight' formal JML specifications for the JavaCard API is reported in [13, 14]. This paper discusses the specification of the `JCSystem` class, which is the class in the API that deals with the JavaCard firewall, (atomic) transactions and transient objects. The `JCSystem` class seems to be the most difficult class in the API to specify, as it cannot be described completely independently of some basic features of the JavaCard Virtual Machine. The class is closely connected with some of the peculiarities of JavaCard as opposed to Java.

This work is part of the EU-sponsored VerifiCard-project which aims to provide formal descriptions of the JavaCard platform and to provide tools for applet verification based on these formal descriptions. (For more details see `http://www.verificard.org`.)

2 JML

JML [8, 9] is a behavioural interface specification language tailored to Java. It can be used to specify Java classes and interfaces by annotating code with invariants and pre- and postconditions of methods, in the style of Eiffel, also known as 'Design by Contract' [10]. JML annotations are a special kind of Java comments: they are preceded by `//@`, or enclosed between `/*@` and `@*/`, so that they are simply ignored by a Java compiler.

Methods can be specified in JML by so-called normal behaviours. These are of the form:

```
/*@ normal_behavior
  @   requires: <precondition> ;
  @    ensures: <postcondition> ;
  @*/
```

Such a 'normal behaviour' specification states that if the precondition holds at the beginning of a method invocation, then the method terminates normally (i.e. without throwing an exception) and the postcondition will hold at the end of the method invocation.

Pre- and postconditions are simply standard Java boolean expressions, extended with several additional operators, for example `\forall` for universal quantification and `==>` for implication. A few more of these additional operations will be explained as we go along.

Java methods can terminate abruptly, by throwing exceptions. If a method can throw an exception, then a more general form of method specification than the normal behaviour above is needed:

```
/*@ behavior
  @   requires: <precondition> ;
  @    ensures: <postcondition>;
  @    signals: (Exception1) <condition1>;
  @    ⋮
  @    signals: (Exceptionn) <conditionn>;
  @*/
```

Such a 'behaviour' specification states that if the precondition holds at the beginning of a method invocation, then the method either terminates normally or terminates abruptly by throwing one of the listed exceptions; if the method terminates normally, then the postcondition will hold; if the method throws an exception, then the corresponding condition will hold.

More than one (normal) behaviour can be specified for a single method. This simply means that the method has to satisfy all of the given (normal) behaviours.

This is a convenient way to specify different cases in the behaviour of a method. The default pre- and postconditions are `requires: true` and `ensures: true`, and these may be omitted in specifications.

In addition to `requires`, `ensures` and `signals` clauses, methods specifications can also include `modifiable` clauses. These clauses specify so-called frame conditions, which say that only certain fields may have their values changed by a method. For example, `modifiable:x` specifies that a method changes only field `x`.

One feature of JML that is of particular importance for this paper is the possibility of using so-called *model fields* (or *model variables*). Model fields are declared in JML annotations and provide specification-only variables: they can not be used in program code, but only in JML assertions. Typically, model fields are introduced to refer to some part of the 'state' encapsulated by objects of a class, a 'piece of state' on which the informal specification implicitly relies, and on which the formal JML specification will explicitly rely. As far as the class `JCSystem` discussed in this paper is concerned, the main difficulty in developing formal JML specifications is introducing appropriate model fields.

Ultimately, there should be a relation between the model variables used in the specification and variables actually used in the implementation, and this relation can again be stated as a JML annotation. Of course, if a method is native, which is the case for many methods of `JCSystem`, there is no concrete implementation to which the models variables can be related.

Normally a model variable and the way its value changes in response to invocations of certain methods is completely described by the specifications of these methods. However, as will be discussed later, for some model variables used in the specification of `JCSystem` this is not the case. The value of some model variables does not only change in response to certain method invocations, but can also change as a side-effect of basic JavaCard language constructs.

3 The JavaCard API

Together with the JavaCard virtual machine (JCVM), the JavaCard API forms the JavaCard runtime environment, or JCRE, as illustrated in Figure 1. The JCVM provides the interpretation of the basic JavaCard language, i.e. of all the JavaCard language constructs. The API is a collection of classes and interfaces providing additional functionality that can be used in JavaCard programs. The JavaCard API (version 2.1.1) [5] comprises 18 classes for exceptions, 16 interfaces for working with cryptographic keys and only 10 fundamental classes (in the package `javacard.framework`). Of these `ISO7816`, `Util`, `Applet` and `Shareable` are elementary from a specification perspective, `PIN` and `OwnerPIN` facilitate working with PIN-codes, and `AID` is a datatype for the identification of applets.

Part of the JavaCard API – namely the classes `APDU` and `JCSystem` – can be understood as an interface to the miniature operating system running on the smart card. The class `APDU` is the interface of the device driver handling

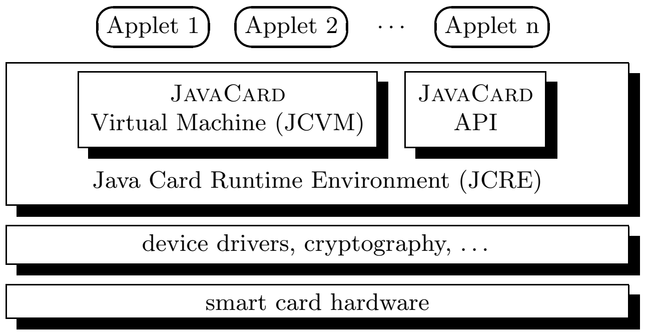
Applet 1
Applet 2
· · ·
Applet n
JavaCard Virtual Machine (JCVM)
JavaCard API
Java Card Runtime Environment (JCRE)
device drivers, cryptography, . . .
smart card hardware

Fig. 1. The JavaCard Platform

communication of the smart card and the card reader, aka the Card Acceptance Device (CAD). A (still incomplete) formal specification for `APDU` is discussed in [14]. The class `JCSystem` provides an interface to the core of the operating system, dealing with the firewall, (atomic) transactions and transient objects. Its specification is the topic of this paper.

A diagram such as Figure 1 is somewhat misleading, as it suggests a clear division between the JCVM and the API, whereas there is a close connection: some parts of the API concern features that are an integral part of the JCVM. It would be more accurate to have some overlap between the API and JCVM boxes in Figure 1. The one class in the API that would be in this overlap is `JCSystem`, as the functionality it provides is intimately connected with the JCVM. A complicating factor in understanding the connection between JCVM and API is that the specification of the JCVM [7] is given at byte-code level, whereas the specification of the API [5] is given at source-code level. The additional JCRE specification [6], which gives the most detailed description of the connection between JCVM and API, uses byte code level in some parts and source code in others.

4 JCSystem

The `JCSystem` class offers the functionality for 3 basic ingredients of the JavaCard platform:

1. the creation of objects in transient memory which are cleared when the applet is deselected or when the card is reset;
2. atomic transactions enabling the undoing of certain (partial) updates when power is lost;
3. the firewall which restricts the access of an applet's objects by other applets unless explicitly granted.

This functionality is embodied in 15 methods (not counting the method `getVersion` which yields the current version number), listed in Figure 2 below. These methods may be considered as the *system calls* of the JavaCard operating system by which applets have access to the basic features of JavaCard. We will discuss the formal specification of transience and the firewall in some detail, and sketch that of atomicity.

As far as the three features listed above are concerned it is hard to draw a line between the API and the JavaCard language (or the JCVM). Unlike the other API classes the class `JCSystem` does not provide a piece of functionality that provides an addition to the bare JCVM and that can be understood in isolation, as the three features are really an integral part of the JavaCard language. This is what makes the specification of `JCSystem` essentially more difficult than the specification of the other API classes. Indeed, the class `JCSystem` is not only specified in Sun's API specification [5], but is also extensively discussed in the JavaCard Runtime Environment (JCRE) specification [6]. A still more detailed description is given by Sun's Java reference implementation of the JavaCard API. In particular, the reference implementation of Sun includes a class `Dispatcher` that does not contain any public fields or methods and is therefore not included in the API specification. It contains the `main` method which performs card initialization (creates fundamental objects like the APDU buffer and installs built-in applets; it is called only once) and card reset (at each power up), and drives the main loop of the card, which processes APDUs resulting in the (de)selection of applets or the dispatching of APDUs to or from the currently selected applet.

The basis for our formal JML specification of `JCSystem` is provided by the three sources of information mentioned above: the (informal) JavaCard API specification [5], the JavaCard Runtime Environment (JCRE) specification [6], and Sun's reference implementation of the API. The following sections discuss the different parts of `JCSystem`, dealing with transient objects, the firewall, and the transaction mechanism, in more detail.

4.1 Transient objects

The JavaCard platform assumes the presence of both persistent memory which retains data even if power is lost, and transient memory which is cleared upon power loss. When objects are created by the `new` operator, they are allocated in persistent memory. The class `JCSystem` provides methods for creating objects in transient memory; these objects are always arrays.

The method

```
makeTransientBooleanArray (short length, byte event)
```

creates (allocates) an array of booleans of the given `length` in transient memory. The array is persistent in the sense that it survives card resets, but its *contents* are cleared (i.e. set to `false`) when the card is reset (e.g. after power loss), or when the applet which created it is deselected. This choice is determined by the parameter `event`, which can be `CLEAR_ON_RESET` or `CLEAR_ON_DESELECT`.

```
public final class JCSystem {

 public static short getVersion();

 /* methods for transient objects */

 public static native byte isTransient(Object theObj);

 public static native boolean[] makeTransientBooleanArray(short length,
                                                           byte event);
 public static native byte[] makeTransientByteArray(short length,
                                                     byte event);
 public static native short[] makeTransientShortArray(short length,
                                                       byte event)
 public static native Object[] makeTransientObjectArray(short length,
                                                         byte event) ;

 /* methods for the transaction mechanism */

 public static native void beginTransaction() throws TransactionException;
 public static native void abortTransaction() throws TransactionException;
 public static native void commitTransaction() throws TransactionException;
 public static native byte getTransactionDepth();
 public static native short getUnusedCommitCapacity();
 public static native short getMaxCommitCapacity();

 /* methods for the firewall */

 public static AID getAID();
 public static AID lookupAID( byte[] buffer, short offset, byte length );
 public static AID getPreviousContextAID();
 public static Shareable getAppletShareableInterfaceObject(AID serverAID,
                                                            byte parameter);
}
```

Fig. 2. The methods of `JCSystem`

The methods `makeTransientByteArray`, `makeTransientShortArray`, `make-TransientObjectArray` are completely analogous. The method `isTransient (Object theObj)` yields a byte with value `NOT_A_TRANSIENT_OBJECT`, `CLEAR_ON_RESET` or `CLEAR_ON_DESELECT` with the obvious meaning.

For a formal specification, we first note that a normal behavior of `make-TransientBooleanArray` requires a restriction on the values of its parameters: $0 \leq \texttt{length} \leq \mathit{free}$ where *free* is the amount of available free transient memory, and $\texttt{event} \in \{\texttt{CLEAR\_ON\_RESET}, \texttt{CLEAR\_ON\_DESELECT}\}$. The firewall imposes additional restrictions. The firewall relies on a partitioning of the object system into separate objects spaces called *contexts*. The JCRE manages a context for each applet, and the firewall restricts access across boundaries between these contexts. Because of the firewall, the normal behavior of `makeTransientBooleanArray` requires that if `event` equals `CLEAR_ON_DESELECT`, the currently active context should equal the currently selected context.

These conditions together constitute the precondition of the normal behavior of the method. Each possible way to negate this precondition represents a precondition of an exceptional behavior, where an exception is thrown. For instance, if $\texttt{length} > \mathit{free}$, a `SystemException` is thrown with reason code `SystemException.NO_TRANSIENT_SPACE`.

The postcondition of `makeTransientBooleanArray`'s normal behavior should express that its result is a non-null transient Boolean array with the correct length, event and contents (viz. all `false`s), that *free* decreases by an amount of `length`[1]. Finally, the JML operation `\fresh` can be used to say that the result is a 'freshly' allocated object. Before writing the actual JML-code of this specification, we should note that the variable *free*, the notions 'currently active context' and 'currently selected context', and the 'event' property of a (transient) object are not directly available when the method is applied. Therefore, we introduce JML model variables for *free* (which we call `_freeTransient`) as well as for the contexts, and assume that each object has an 'event' model field telling whether that object is `NOT_A_TRANSIENT_OBJECT`, or else `CLEAR_ON_RESET` or `CLEAR_ON_DESELECT`.

We must make sure when verifying the API and applets that the values of these model variables and properties are properly maintained. For instance, the increase and decrease of the run-time stack of the JCVM will influence `_freeTransient`. The 'currently active context' can change with a method call, and is (in)directly registered in the JCVM run-time-stack. The 'currently selected context' changes with the (de)selection of applets by the dispatcher. In some cases (the changes in) the values of model variables and properties can be maintained in the JML-specification itself. For instance, one could imagine that the value of the 'currently selected context' is maintained in the `select`- and `deselect`-methods of the `Applet`-class. Otherwise, the maintenance has to be built in into the semantics of the relevant JavaCard statements.

[1] Here we possibly oversimplify things a bit, e.g. we ignore any fixed overhead to record the length of the array.

```
public static native boolean[] makeTransientBooleanArray(short length,
                                                          byte event)
throws SystemException;

/*@ normal_behavior
  @   requires: 0 <= length && length <= _freeTransient &&
  @             (event == CLEAR_ON_RESET || event == CLEAR_ON_DESELECT) &&
  @             (event == CLEAR_ON_DESELECT
  @                              ==> _selectedContext == _activeContext);
  @ modifiable: _freeTransient;
  @    ensures: \result != null && \result.length == length &&
  @             \result._event == event && \fresh(\result) &&
  @             _freeTransient == \old(_freeTransient) - length &&
  @             \forall (byte i) 0 <= i < length ==> \result[i] == false;
  @ also
  @ behavior
  @    signals: (SystemException e)
  @            (e.getReason() == SystemException.ILLEGAL_VALUE &&
  @             (length < 0 ||
  @              (event != CLEAR_ON_RESET && event != CLEAR_ON_DESELECT)))
  @           ||
  @            ( e.getReason() == SystemException.NO_TRANSIENT_SPACE &&
  @             _freeTransient < length)
  @           ||
  @            (e.getReason() == SystemException.ILLEGAL_TRANSIENT &&
  @             event == CLEAR_ON_DESELECT &&
  @             _selectedContext != _activeContext);
  @*/
```

Fig. 3. JML specification of `makeTransientBooleanArray`

We will not resolve this issue here, and just assume that the appropriate (model) variables and properties are available and properly maintained. By conventions, their names are distinguished by an initial underscore.

This then results in the JML-specification for `makeTransientBooleanArray` in Figure 3.

The JavaCard documentation does not specify the behavior in the case where `length` < 0. Whatever class declares `_freeTransient` should specify the invariant $0 <=$ `_freeTransient` (possibly also giving an upper limit by means of a constant `_MAX_SIZE_OF_TRANSIENT`). The question may be raised whether it is necessary to specify that the allocation is not actually done in persistent memory (as is required by the informal specification), or even that the allocation does not use memory which is already allocated to other objects.

The actual clearing of transient arrays should be specified in the dispatcher. This may call for a detailed administration of all created transient arrays, necessitating an extension of our specification. For instance, we might use a model variable `_transientMemory` modelling the whole transient memory and specify the precise allocation in the ensures-clause of the normal behavior (and make _freeTransient a field of _transientMemory). Alternatively, one could model a separate transient memory for 'clear_on_reset' arrays and separate transient memories for 'clear_on_deselect' arrays for each possible applet, identified by its AID.

The specification of isTransient is rather trivial:

```
public static native byte isTransient(Object theObj);

/*@  behavior
  @     ensures: \result == theObj._event;
  @*/
```

4.2 The firewall

The firewall mechanism prohibits an applet's access to objects owned (created) by other applets. The rules as detailed in the JavaCard Runtime Environment (JCRE) specification are quite elaborate. The main principles are that the JCRE can access any object, but applets can only access objects owned by applets in the same package, objects designated as JCRE entry-point objects, and 'shareable' objects to which access is explicitly granted by their owners.

Applets have a certain context, which is basically the package they belong to, and the JCRE is considered to have no context. As long as an applet is selected, that applet's context is the currently selected context. When a method of an applet is called, that applet's context becomes the currently active context. There is also a 'previous context' which is the context of the applet which called the currently active method, possibly via intermediate JCRE methods.

JCSystem provides a method

```
public static Shareable getAppletShareableInterfaceObject
                              (AID serverAID, byte parameter)
```

by which an applet may ask permission to access an object owned by the applet identified by the given serverAID. It follows from the informal API specification [5] (from the specifications of getAppletShareable-InterfaceObject of the class JCSystem and of getShareableInterface-Object of class Applet, to be precise) that this method calls the method getShareableInterfaceObject of the applet identified by serverAID, passing it the AID of the calling applet (or null if the caller is a JCRE-method) and the given parameter. The AID of the calling applet can be obtained using the method getPreviousContextAID. If the serverAID is invalid, null is returned. Therefore, if the server is valid, and the caller is valid (or the JCRE), the specification of getAppletShareableInterfaceObject will be that of the server's getShareableInterfaceObject.

This is formalized in the JML-specification for getAppletShareable-InterfaceObject in Figure 4. It requires the introduction of more model variables, as discussed below.

```
public static Shareable getAppletShareableInterfaceObject(AID  serverAID,
                                                           byte parameter)
/*@ normal_behavior
  @   requires: _previousContext == _jcreContext &&
  @             _registeredAIDs.has(serverAID);
  @    ensures: \result ==
  @             (_appletTable.apply(serverAID)).getShareableInterfaceObject
  @                                           (null,parameter);
  @ also
  @ normal_behavior
  @   requires: _previousContext != _jcreContext &&
  @             _registeredAIDs.has(serverAID);
  @   ensures: \result ==
  @            (_appletTable.apply(serverAID)).getShareableInterfaceObject
  @                                    (getPreviousContextAID(),parameter);
  @ also
  @ normal_behavior
  @   requires: !_registeredAIDs.has(serverAID);
  @    ensures: \result == null;
  @*/
```

Fig. 4. JML specification of `getAppletShareableInterfaceObject`

The model variable `_previousContext` is akin to the model variables `_selectedContext` and `_activeContext` introduced in Section 4.1. The `_selectedContext` is set and reset when applets are selected and deselected. The other two have to be maintained as part of the semantics of the method call. In fact, their values could be extracted from the run-time stack. The related model constant `_jcreContext` represents the context of the JCRE; it is different from any applet's context.

Two model variables used in the specification of `getAppletShareable-InterfaceObject` are of more complicated types than the model variables seen so far. First, there is a model variable `_registeredAIDs`, which is the set of all the AID's of installed applets. Second, there is a model variable `_appletTable`, which is a partial function from AID's to applets, that, given an AID, returns the applet with that AID, if such an applet is installed. So the domain of `_appletTable` will be `_registeredAIDs`, and this could in fact be included as an invariant. The values of these model variables will change in response to invocations of the method `register` of an applet.

In specifications one often needs model variables that represents sets and functions, such as `_registeredAIDs` and `_appletTable` above. For this reason the JML distribution comes with a package `edu.iastate.cs.jml.models` that implements many mathematical notions that are frequently needed in specifications. This package provides suitable classes for `_registeredAIDs` and `_appletTable`:

```
  public model JMLObjectSet         _registeredAIDs;
  public model JMLObjectToObjectMap _appletTable;
```

The method `has` for `JMLObjectSet` and `apply` for `JMLObjectToObjectMap` used in the specification of `getAppletShareableInterfaceObject` in Figure 4 have the obvious interpretations. A detailed description of these classes is given as part of the JML release that can be obtained at `http://www.cs.iastate.edu/~leavens/JML.html`.

Sun's reference implementation of the JavaCard API provides a particular representation of the information in `_registeredAIDs`, `_appletTable`, and `_previousAID`. In fact, it uses an array of objects `theAppletTable` to record the mapping `_appletTable`. Internally, the reference implementation does not use AIDs but indexes in this array to identify applets. To ensure that the specification is in agreement with this implementation one would use the standard technique of establishing an invariant that expresses the correspondence between the abstract model variable and the concrete implementation.

The method `lookupAID`, by which an applet can obtain `AID`s to pass to `getAppletShareableInterfaceObject` accesses the internal applet table in much the same way as `getAppletShareableInterfaceObject` and is easy to specify.

The two remaining methods related to the firewall, `getAID` and `getPreviousContextAID`, return the AIDs associated with the current and the previous applet context, respectively. They could be specified by introducing model variables `_currentContextAID` and `_previousContextAID`, and then specifying

```
 public static AID getPreviousContextAID()

 /*@ normal_behaviour
   @  ensures: \result == _previousContextAID;
   @*/
```

and similarly for `getAID`. But observe that there is then effectively no difference between `getPreviousContextAID()` and the method variable `_previousContextAID`. One could simply do away with the model variable and use the method instead. (Indeed, the specification of `getAppletShareableInterfaceObject` in Figure 4 already uses the method invocation `getPreviousContextAID()` rather than some model variable `_previousContextAID`.) Essentially the methods `getAID` and `getPreviousContextAID` are too fundamental for an interesting specification in JML.

Something interesting that could still be specified is the relation between contexts and AIDs. A model variable for this relation could be introduced in order to express the invariants that hold between `_currentContext` and `_currentContextAID` and `_previousContext` and `_previousContextAID`.

4.3 Atomic Transactions

All updates of single array elements, object fields and class fields in persistent memory are atomic. If a failure or power loss occurs during an update, the field is restored to its previous value. With modern hardware it is not difficult to obey this requirement, and in formal specifications atomic updates are in fact silently assumed. The `Util`-class of the JavaCard API provides methods for atomic and non-atomic updates of arrays.

For more complicated updates, the `JCSystem`-class provides the methods `beginTransaction()`, `abortTransaction()` and `commitTransaction()`. A (hidden) variable `transactionDepth`, which is 0 initially, is incremented by `beginTransaction()`, decremented by `abortTransaction()` and `commitTransaction()`, but should always have a value of 0 or 1, otherwise a `TransactionException` is thrown. This outlines a behavior specification of these methods.

Essentially, `beginTransaction()` should create a 'backup' of persistent memory in a second persistent memory, and `abortTransaction()` should 'restore' it. However, this is not feasible as the amount of persistent memory is severely restricted and writing to it is expensive. Therefore, a *commit buffer* is provided, the size of which is implementation-dependent. The use of this buffer may follow different schemes (as outlined in [12]), of which we just choose one ('old value logging', as opposed to 'new value logging') for our specification.

For each update of a byte of persistent memory, if `transactionDepth` $>$ 0, the value to be overwritten is recorded in the commit buffer, provided that no value is yet recorded for that byte. The buffer is cleared by `commitTransaction()`, but `abortTransaction()` restores the old values. Also, `abortTransaction()` is implicitly called when the JCRE regains control, upon applet deselection, card reset (e.g. after a power loss), or any kind of failure. This may happen even if `abortTransaction()` is in progress. Consequently, from the perspective of the applets, the result of such a transaction is always consistently the collection of new values or the collection of old values. The atomicity of transactions ultimately relies on the fact that the decrease of `transactionDepth` is atomic, and the last action of `abortTransaction()`.

Moreover, any object created during a transaction, either in persistent or in transient memory, is deleted and has its memory freed upon `abortTransaction()`. Each reference to such an object, even on the run-time stack, is set to `null`.

In the specification we introduce a (rather complicated) model variable `_theCommitBuffer`. We assume that the semantics of updates and object creation is extended so as to include the recording of old values and pointers in this model object. The specification of `abortTransaction()` will state that the values of all object fields and array elements recorded in the commit buffer equal those recorded, that any references to newly created objects are set to `null` and that the memory of these objects is freed.

The specification of three other methods dealing with atomic transactions, `getTransactionDepth`, `getUnusedCommitCapacity` and `getMaxCommit-`

Capacity are quite straightforward. The latter two extract properties from _theCommitBuffer.

5 Related Work

Most of the work on formalising the JavaCard platform has focussed on the JCVM rather than the API. Still, some (formal) models of transaction mechanism, firewall, and transient memory have been given. For example, transaction mechanisms are described in [15] (using B), [12] and [3] (using Z), and the firewall in [11] (using B). As mentioned earlier, as far as firewall, atomic actions, and transient memory are concerned it is hard to draw a line between the API and JCVM, so this work is not unrelated to what we have done. The model variables we need in our JML specifications should have counterparts in formal descriptions of the JCVM, as explained in more detail below.

6 Conclusion

Although some details of the specification of the JavaCard API are somewhat subtle, the specification as a whole turns out to be rather small. All methods have a specification which is not essentially larger than that of makeTransientBooleanArray as given in Figure 3.

The model variables that have to be introduced in our JML specifications make explicit many of the informal notions used in the existing informal documentation. This can help to clarify and improve these informal specifications.

Normally, the values of model variables and the way these change in response to methods invocations is completely described by the JML specifications. However, for many model variables used in the specification of JCSystem this is not the case. Maintaining the correct values of these model variables is an integral part of the semantics of normal JavaCard statements. For example, any assignment to a persistent object field that occurs during a transaction affects _theCommitBuffer, so the semantics of assignment should include this side-effect on _theCommitBuffer. Or, to give another example, the variable _activeContext may need to be changed at every method invocation, so the semantics of method invocation should include a side-effect on _activeContext. Note that a comprehensive account of all such examples comes down to a specification of the differences between Java and JavaCard.

All this means that ultimately a specification of the JavaCard API cannot be considered on its own, but has to be considered together with a formalisation of the JavaCard language itself, e.g. a formal description of the JCVM or —in our case— our denotational semantics of JavaCard in PVS. The 'side-effects' mentioned above should then be made part of the implementation of certain virtual machine instructions (e.g. *invoke* bytecodes) c.q. be included in the semantics of JavaCard source code statements. The list of model variables needed in a formal JML specification gives a good overview of the variables that have to be maintained by a JCVM in order to, say, implement the firewall.

When verifying JavaCard source code using the LOOP tool and PVS, verifying any properties that depend on (the model variables of) `JCSystem` will require some mechanism by which (JML) model variables can be associated with the external operations they are subject to. How this should be accomplished is a matter of further investigation. Note that this issue is not particular to our approach to verification using the LOOP tool and PVS: whenever one wants to adapt an existing approach of Java-verification to JavaCard the question of how to deal with the peculiarities of JavaCard as opposed to Java arises.

References

1. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools ans Algorithms for the Construction and Analysis of Software (TACAS)*, number 2031 in LNCS, pages 299–312. Springer, Berlin, 2001.
2. J. van den Berg, B. Jacobs, and E. Poll. Formal Specification and Verification of JavaCard's Application Identifier Class. In I. Attali and T. Jensen, editors, *Proceeding of the first JavaCard Workshop (JCW'2000)*, LNCS. Springer Verlag, 2001. To appear.
3. P. H. Hartel, M. J. Butler, E. de Jong, and M. Longley. Transacted memory for smart cards. In *10th Formal Methods for Increasing Software Productivity (FME)*, LNCS. Springer Verlag, 2001.
4. P. H. Hartel and L. A. V. Moreau. Formalizing the safety of Java, the Java virtual machine and Java Card. *ACM Computing Surveys*, 2001. to appear.
5. *The Java Card 2.1.1 Application Programming Interface (API).* Sun Microsystems, 2000.
6. *The Java Card 2.1.1 Runtime Environment (JCRE) Specification.* Sun Microsystems, 2000.
7. *The Java Card 2.1.1 Virtual Machine (JCVM) Specification.* Sun Microsystems, 2000.
8. G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov and B. Rumpe, editors, *Behavioral Specifications of Business and Systems*, pages 175–188. Kluwer, 1999.
9. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Techn. Rep. 98-06, Dep. of Comp. Sci., Iowa State Univ. (`http://www.cs.iastate.edu/~leavens/JML.html`), 1999.
10. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2^{nd} rev. edition, 1997.
11. Stéphanie Motré. Formal model and implementation of the Java Card dynamic security policy. Technical Report SM-99-09, Gemplus Research Lab, 1999. Presented at AFADL'2000.
12. Marcus Oestreicher. Transactions in Java Card. In *15th Annual Computer Security Applications Conference (ACSAC'99)*, pages 291–298. IEEE, 1999.
13. E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000)*, pages 135–154. Kluwer Acad. Publ., 2000.
14. E. Poll, J. van den Berg, and B. Jacobs. Formal Specification of the JavaCard API in JML: The APDU class. *Computer Networks*, 2001. To appear.

15. Denis Sabatier and Pierre Lartigue. The use of the B formal method for the design and the validation of the transaction mechanism for smart card applications. *Formal Method in System Design*, 17(3):145–272, 2000. Special issue on FM'99.