Fully Verified Java Card API Reference Implementation

Wojciech Mostowski

Computing Science Department, Radboud University Nijmegen, the Netherlands
woj@cs.ru.nl

Abstract. We present a formally verified reference implementation of the Java Card API. This case study has been developed with the KeY verification system. The KeY system allows us to symbolically execute the Java source code of the API in the KeY verification environment and, in turn, prove correctness of the implementation w.r.t. formal specification we developed along the way. The resulting formal API framework (the implementation and the specification) can be used to prove correctness of any Java Card applet code. As a side effect, this case study also serves as a benchmark for the KeY system. It shows that a code base of such size can be feasibly verified, and that KeY indeed covers all of the Java Card language specification.

1 Introduction

We present a formally verified reference implementation of the Java Card API version 2.2.1 [11]. Java Card is a technology used to program smart cards and it comprises a subset of the desktop Java; a subset of the programming language itself, and a cut down version of the API. Reference implementations of the API that are normally available for Java Card are developed for a particular running environment, usually a Java Card simulator or emulator. Some of the API routines are not implementable in Java alone. For example, Java Card transaction mechanism functionality is normally provided by the Java Card VM and/or the operating system of the actual smart card hardware. A simulator provides similar functionality through Java Native Interface (JNI). Thus, the API implementation consists of the Java part and the JNI part, the latter reflecting the low-level behaviour of the card. Both parts together enable the API implementation to run on a simulator. In contrast, our API implementation has been developed for the KeY interactive verification system[1] [1]. That is, the implementation is designed to be symbolically executed in a formal verification environment. Similarly to a simulator, the KeY verifier also needs to handle low-level Java Card specific routines. In this case the JNI functionality is provided by the formal model of the Java Card environment embedded in the KeY verifier logic; a set of specialised logic rules to symbolically execute Java Card native method calls. Figure 1 shows corresponding architectures of the API implementations.

Along with the implementation we developed a set of formal specifications for the API. And, naturally, we used the KeY system to prove the correctness

[1] http://www.key-project.org

of our implementation w.r.t. to the specification. The proving is done by means of symbolic execution of the JAVA source code of the API implementation in the KeY system and then evaluating the specification formulae on the resulting execution state.

This case study serves three main goals: (i) an API framework (implementation and specification) for verification of JAVA CARD applet source code, (ii) consistency of the informal API specification [5], and (iii) as a benchmark for the KeY system and as a verification case study itself that explores the usability of formal methods in practice. We elaborate on these goals in the following paragraphs.

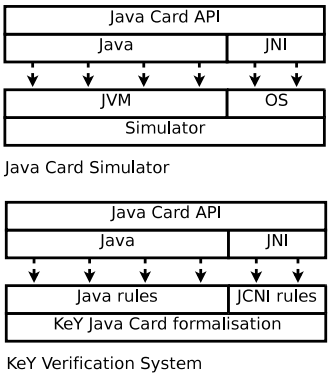

Fig. 1. Architecture of JAVA CARD API implementations

In the current PinPas JAVA CARD project[2] we investigate fault injection attacks on smart cards. A possibility for a fault injection calls for appropriate countermeasures. One of such countermeasures are simply modifications to JAVA applet source code to detect and neutralise faults. Such modifications can result in a complex code. One of our goals in the project is to be able to formally verify such modified source code, i.e., that the more complex fault-proof code behaves the same way as the simple fault-sensitive code, as described in earlier work [4]. For this we need a verification tool that faithfully reflects JAVA CARD programming language semantics, and also a faithful reflection of the API behaviour. The verification tool of our choice, the KeY system, already provides the formalisation of the whole of JAVA CARD programming language semantics. We said earlier that JAVA CARD is a subset of JAVA. In practice, because of the specifics of the smart card technology, JAVA CARD introduces additional complications to the language. Namely, two different kinds of writable memory (persistent EEPROM and transient RAM), an atomic transaction mechanism, and applet firewall mechanism. All these features are embedded into the JAVA CARD Virtual Machine (JCVM) running on a card. In effect, this sometimes changes the semantics of the primitive JAVA statements in the context of a smart card application. The KeY system already supports all of the mentioned JAVA CARD specific features [1, Chapter 9], with a notable exception of the firewall mechanism, which is being integrated into KeY in parallel to this work.

When it comes to the API behaviour, however, our approach so far was to specify and implement only those API methods that are required for a given verification task at hand [1, Chapter 14]. Ultimately, to be able to verify *any* given JAVA CARD applet code w.r.t. wide range of properties, the specification *and* the implementation for the whole of the API should be present. The need

[2] `http://www.win.tue.nl/pinpasjc/`

for having *both* the specification and implementation is justified as follows. First of all, reasoning on the level of interfaces, i.e., relying on method contracts only, is not sufficient for some properties. In particular, strong invariant properties require the evaluation of all intermediate atomic states of execution, including the ones that result from the execution of an API method. Moreover, in principle, method contracts cannot be applied in the context of Java Card transactions. Here, sometimes a special "transaction aware" contract is needed, which in some cases can only be constructed by hand based on the actual implementation of the method. In essence, one needs to state the behaviour of the method in terms of conditionally updated data, rather than the actual data. For some API methods that interact heavily with the transaction mechanism giving a suitable contract is not possible at all [1, Chapter 9]. Finally, in some few cases, formal reasoning based on the code instead of the specification can be simply more efficient.

We tried to make the reference implementation as faithful as possible, i.e., closely reflecting the actual card behaviour. The official informal specifications were closely followed, and sometimes also the actual cards were tested to clarify the semantics of some API methods. However, some Java Card aspects had to be omitted in the implementation and it is obvious that certain gaps will always remain between our reference implementation executing in a formal verification environment and API implementations executing on actual smart cards. We discuss those issues in detail in Section 3.

One of the other goals of the PinPasJC project is to discover inconsistencies between the actual cards and the official informal Java Card specifications, i.e., to simply find implementation bugs on the cards. Implementing the whole of the Java Card API gave us a good chance to review the informal specification and identify "hot spots" – descriptions that are likely to be ignored or misinterpreted, unintentionally or on purpose. We mention this briefly in Section 4.

Finally, our reference implementation serves as a verification case study. First of all, it gives a general idea of how big a code base can be subjected to formal verification by the KeY tool. The Java Card API consists of 60 classes adding up to 205KB of Java source code (circa 5000 LoC) and 395KB of formal specifications (circa 10000 LoC). The KeY system managed to deal with it giving satisfying verification results, although not without problems, see Section 4. Moreover, the case study shows the compliance of the KeY system to the Java Card language specification – the reference implementation utilises practically all of the Java Card language subset and was successfully verified by the KeY system. Last, but not least, this case study will serve to further optimise and tune the KeY system in terms of performance (resource-wise) and minimising user interaction.

Related Work. The work we present here is one of many that formally treats the Java Card API library. However, to the best of our knowledge, we are the first

ones to give a relatively full, (soon) publicly available reference implementation of version 2.2.1 of the API together with specifications. The older JAVA CARD development kits distributed by Sun contained a reference implementation of the API up to version 2.1.1.[3] Since that version the source code of the API is no longer available with the development kits. Our code borrows ideas from Sun's reference implementation, there are however two major differences to be noted. First, our implementation treats the newer API version which introduces many new features. Secondly, our back-end system is the KeY JAVA CARD model, while Sun's API is implemented for the JCWDE simulator.

When it comes to formal treatment of the API, a full set of JML specifications[4] [9] for the API version 2.1.1 has been developed for the LOOP verification tool [6] and ESC/JAVA2 [2]. These efforts, however, do not include the implementation itself, only specifications. As a side effect of our work, we also constructed lightweight JML specifications of the API version 2.2.1 for ESC/JAVA2.[5] Moreover, in an earlier work we investigated specification of the JAVA CARD API in OCL [7].

Recently a work has been published on a method to formally verify native API implementations based on specification refinement [10]. Three levels of specifications for the native API code are given in the Coq language: functional specification, high-level description, and low-level description. The last level is not yet the actual implementation of the code on the card (normally written in C or even assembly), but is claimed to have a one to one correspondence with the code running on the card. The correctness of the low-level description is verified by means of refinement relation between the three levels of specification. The main goal is to verify the actual API implementations found on cards, while we aim at providing a JAVA source code verification framework to be used *outside* of the card.

Structure of the Paper. The rest of this paper is organised as follows. Section 2 describes tools and methodologies we used: the KeY system, its logic, the JAVA CARD formal model on top of which our reference implementation was written, and a discussion about the choice of the specification language. Section 3 describes the implementation and its specification in more detail with samples of both, while Section 4 gives some insights into the verification effort and discusses our experience. Finally, Section 5 concludes the paper.

2 Tools and Methodology

In this section we describe the main building blocks of our case study: the KeY system, the KeY model of the JAVA CARD environment, and briefly the JAVA CARD Dynamic Logic, which we used a specification language.

[3] `http://java.sun.com/products/javacard/dev_kit.html`
[4] `http://www.sos.cs.ru.nl/research/escjava/esc2jcapi.html`
[5] Available at `http://www.cs.ru.nl/~woj/software/software.html`

2.1 Verification Tool

The verification tool of our choice for this case study is the KeY system. The KeY system is a highly automated interactive program verifier for Java and Java Card programs. Currently KeY supports verification of sequential Java without floating point arithmetic and dynamic class loading, and practically all of Java Card[6] including the Java Card transaction mechanism and (non)persistence of the Java Card memory [1, Chapter 9]. The formalism behind the KeY system is the Java Card Dynamic Logic (Java Card DL) [1, Chapter 3] built on top of first order logic. The rules of the logic are used to symbolically execute Java programs in the KeY prover and then evaluate the properties to be verified. We describe this idea with a simple example. Take the following Java code:

```
public void decreaseCounter() { if(counter > 0) counter--; }
```

What we would like to specify and prove is that assuming the counter is non-negative it will stay non-negative after the execution of the method, i.e., that the method preserves the invariant `counter >= 0`. A corresponding Java Card DL formula would take the following form (we use the actual KeY system syntax):

```
self != null & self.counter >= 0 ->
  \<{ self.decreaseCounter(); }\> self.counter >= 0
```

The formula itself does not contain any class or method definitions, these are implicitly present in the prover. The left side of the implication (`->`) represents the state in which we are about to execute the program, i.e., the precondition of the program. The program itself, calling of a method `decreaseCounter` on the object `self`, is included in the diamond modality `\<·\>`. The formula attached to the modality is to be evaluated after the program in the modality executes, thus, the formula represents the postcondition of the program. In Java Card DL the diamond modality requires the program to terminate and do so in a non-abrupt fashion. In particular, the program is not allowed to throw any exceptions. Contrary to the diamond modality, the box modality `\[·\]` does not require (non-abrupt) termination. In our work, however, the stronger (for deterministic programs) diamond semantics is always used.

The first few simplification steps transform the formula above into a Java Card DL sequent:

```
self != null, self.counter >= 0 ==>
\<{ self.decreaseCounter(); }\> self.counter >= 0
```

The left side of the sequent (marked by `==>`), the antecedent, represents the assumptions and the right side, the succedent represents the proof goal. The next few rules of the symbolic execution unfold the sequent into two branches:

[6] With a notable exception of the Java Card firewall mechanism. However, this does not limit the set of Java Card programs that can be verified with KeY, only the set of properties, see Section 3 for details.

```
self != null, self = null, self.counter >= 0 ==>
\<{ throw new NullPointerException(); }\> self.counter >= 0

self != null, self.counter >= 0 ==>
\<{ if(self.counter > 0) self.counter--; }\> self.counter >= 0
```

The first branch represents the null value check for accessing the object `self`. Whenever an object is accessed (field access or method call), the JAVA CARD DL calculus establishes non-nullness of the referenced object which, if not satisfied, would cause a null pointer exception. This branch is easily closed (proved) by a contradiction in the antecedent `self = null` and `self != null`. We skip further null pointer checks in the rest of the example.

The JAVA CARD DL rule for the `if` statement splits the second sequent into two branches that correspond to the execution paths of the `if` statement:

```
self != null, self.counter >= 0, self.counter > 0 ==>
\<{ self.counter--; }\> self.counter >= 0

self != null, self.counter >= 0, self.counter <= 0 ==>
\<{ }\> self.counter >= 0
```

The second branch does not contain any program in the modality anymore, the modality is removed and the postcondition can be evaluated to true based on the assumptions. The first branch contains an assignment. Applying a corresponding JAVA CARD DL rule results in a state update of the program under execution, which is represented in the following way:

```
self != null, self.counter >= 0, self.counter > 0 ==>
{self.counter := self.counter - 1}\<{ }\> self.counter >= 0
```

JAVA CARD DL state updates are very much like JAVA assignments, except that they are only allowed to appear in their canonical form, in particular, the right hand side of an update has to be side effect free. Further steps in the proof remove the empty modality from the sequent and apply the state update on the formula that is attached to it:

```
self != null, self.counter >= 0, self.counter > 0 ==>
self.counter - 1 >= 0
```

This sequent is easily closed (proved) by simple integer arithmetic reasoning.

Obviously, this simple example cannot discuss all of the details of the logic, in particular, how dynamic method binding is done or how object aliasing is handled by the state update mechanism. All the details can be found in [1, Chapter 3]. However, the last thing we want to discuss here is the treatment of exceptions. The diamond modality requires that no exceptions are thrown. Nevertheless, one can easily construct a proof obligation stating that a method is allowed to throw a given kind of exception by simple program transformation:

```
exc = null & ... -> \<{ try { self.decreaseCounter(); }
  catch(SomeException e) { exc = e; } }\> self.counter >= 0
```

Here, `SomeException` possibly thrown by method `decreaseCounter` is caught by the `try-catch` block resulting in a non-abruptly terminating program. Moreover, it is possible to distinguish the abrupt termination state in the postcondition by making a case distinction on the value of `exc`. A non-null value of `exc` determines that an exception indeed occurred. This exception treatment is essential to how exceptions are treated when higher level specification languages are translated into JAVA CARD DL. The KeY system provides interfaces for both JML [8] and OCL [14]. A specification written in JML or OCL together with the associated code is automatically translated into JAVA CARD DL and then can be verified with the KeY prover. If the specification happens to state exceptional behaviour, e.g., with JML's `signals` clause, the mechanism described above is used during translation.

2.2 JAVA CARD Native Interface

Each JAVA CARD API implementation relies on a native interface to the underlying smart card execution environment (actual hardware or a simulator). Our implementation is meant to be symbolically executable in the KeY system. Thus, the native code interface has to be provided by KeY itself. For this purpose, we equipped the KeY system with a dedicated JAVA class with a number of JAVA CARD specific native methods. As a convention all such methods are named with a `jvm` prefix. Here is an excerpt from the `KeYJCSystem` class:

```
public static native byte jvmIsTransient(Object theObj);
public static native byte[] jvmMakeTransientByteArray(
  short length, byte event);
public static native void jvmBeginTransaction();
public static native void jvmCommitTransaction();
public static native void jvmArrayCopy(byte[] src, short srcOff,
  byte[] dest, short destOff, short  length);
```

Whenever the KeY system encounters a call to one of these methods an axiomatic JAVA CARD DL rule is used to reflect the result of execution in the KeY verifier. For example, a call like this:

```
\<{ transType = KeYJCSystem.jvmIsTransient(obj); ... }\> ...
```

results in a state update of the following form:

```
{transType := obj.<transient>}\<{ ... }\> ...
```

Here `<transient>` is an implicit attribute associated with each object in the KeY JAVA CARD model that indicates whether a given object is persistent (kept in card's EEPROM) or transient (kept in RAM). For example, the code resulting from the execution of `jvmMakeTransientByteArray` sets this attribute to `event` in the array that is being created. The value of `event` indicates on which event (card reset or applet deselection) the contents of the transient array should be cleared.

All of the native methods declared in the `KeYJCSystem` class have such corresponding JAVA CARD DL axiomatic rules. Then it is possible to give the reference implementation of the JAVA CARD API in terms of this native interface, for example:

```
public class JCSystem {
  public static byte isTransient(Object theObj){
      if(theObj == null) return NOT_A_TRANSIENT_OBJECT;
      return KeYJCSystem.jvmIsTransient(theObj); }

  public static byte[] makeTransientByteArray(short length, byte event)
    throws SystemException, NegativeArraySizeException {
      if(event != CLEAR_ON_RESET && event != CLEAR_ON_DESELECT)
        SystemException.throwIt(SystemException.ILLEGAL_VALUE);
      if(length < 0) throw KeYJCSystem.nase;
      return KeYJCSystem.jvmMakeTransientByteArray(length, event); } }
```

2.3 Specification Language

The KeY system supports specifications written in JML or OCL. OCL is not best suited for a case study like this one, it is relatively high level and not too closely coupled to JAVA [7]. A perfect solution would be to use JML, which provides a specification language closely related to JAVA. Moreover, large parts of existing JML specifications for JAVA CARD API [9] could be reused. JML too, however, currently poses one major problem for this case study, namely, the semantics of the generated proof obligations (or rather method contracts associated with a given class and method specification), and the current inability of KeY to manipulate easily the way the contracts are generated. Without going into too much detail, we want our contracts to preserve invariants for the objects of our choice. In most cases this is simply the object a given method is invoked on, and possibly objects passed as parameters or stored in instance attributes. Currently the KeY system does not allow such fine grained selection of object invariants when generating proof obligations from JML specifications.

To solve this problem, we used JAVA CARD DL itself as a specification language, i.e., provided readily generated JAVA CARD DL contracts customised to our needs. This approach does not introduce any complication into the process of constructing specifications. The semantics of JAVA CARD DL expressions and the actual syntax is very close to those of JML. The main difference is that a JAVA CARD DL specification already constitutes a full method contract, thus, one has to manually specify which invariants for which objects are to be included in the precondition and the postcondition of the method. For example, suppose we have the following class with JML annotations:

```
public class MyClass {
  int a=0; //@ invariant a >= 0;
  /*@ requires val >= 0; ensures a = val; assignable a; @*/
  void setA(int val) { a = val; } }
```

Then, assuming that we want to establish preservation of the invariant only for one instance of the class (`self`), the corresponding JAVA CARD DL contract takes the following form:

```
MyClass_setA_contract { \programVariables { MyClass self; int val; }
  self.a >= 0 & val >= 0 ->
    \<{ self.setA()@MyClass; }\> (self.a = val & self.a >= 0)
  \modifies { self.a } };
```

One more advantage of specifying contracts directly in JAVA CARD DL is the possibility to take "shortcuts". For example, one can directly specify the persistency type of an object by referring to its `<transient>` attribute. In JML that would require including the `isTransient` method call in the specification. Such shortcuts improve considerably on the size of the resulting proofs.

Our approach of considering invariants for single object instances assumes that changes to one instance of an object cannot influence the invariant of another instance. That is, we assume there is no inter-object data aliasing. To get confidence that this is indeed the case we would also have to prove that data is properly encapsulated within objects. Currently we cannot do this in KeY in a simple way, proof obligations for proving encapsulation have to be created manually [1, Section 8.5.2]. For a case study of this size this is infeasible. It is of course also possible to employ other formal techniques to prove data encapsulation, for example data universes [3]. On the other hand, for the JAVA CARD API this is not a big issue. Our implementation hardly ever copies data by reference and declares most of the relevant data private, which prohibits direct violation of other objects' invariants.

Finally, we should mention that the KeY JML front-end undergoes heavy refactoring at the moment (partly because of the described deficiencies). Once complete, verification based on JML version of our specification should be possible.

3 Implementation and Specification of the API

The JAVA CARD API [11, 12] provides an interface for smart card specific routines. It is relatively small (60 classes and interfaces) and does not really share common features with the regular desktop JAVA API. Only the very basic classes (like `Object` and `NullPointerException`) are present in both APIs. Apart from that the JAVA CARD API version 2.2.1 provides support for the following smart card specific features: JAVA CARD applets, APDU (Application Protocol Data Units) communication, AID (Applet IDentifiers) registry lookup, owner PIN objects, the atomic transaction mechanism, JAVA CARD inter applet object sharing through the JAVA CARD applet firewall, the JAVA CARD Remote Method Invocation (RMI) interface, cryptographic keys and ciphers, and simple JAVA CARD utility routines. The specifics of the JAVA CARD platform requires the API to have

a small memory footprint. Thus, Java Card does not support strings and associated classes, collections, etc. Moreover, most of the classes that are present in the API are modified to enable low resource usage. For example, cryptographic routines are implemented with a smaller amount of interfaces and methods (compared to Java Cryptography Extensions) and operate only on byte arrays.

Our reference implementation follows the official documentation as closely as possible. However, implementation of some features would be very difficult and the amount of work required would not compensate for the possible gains. Moreover, an over-engineered implementation would be very difficult to verify. Another reason for leaving out certain features is the inability to formally reason about them in KeY.

The first item on the unimplemented feature list are the cryptographic routines. Giving a functional implementation of ciphers and keys in Java that would be easy to understand and verify is simply infeasible. In fact, actual smart cards incorporate a cryptographic coprocessor and highly optimised native code is used for the implementation of ciphers and keys. Thus, our implementation does not contain any actual cryptographic routines. However, all the other features of the cipher and key classes are implemented. For example, the lengths of encryption blocks depending on the encryption algorithm are accurately calculated, or `CryptoExceptions` are reported on all conditions that do not involve checking the result of cryptographic calculations, e.g., that the key is initialised or that the plain text does not exceed its maximum allowed length.

The second unimplemented feature is the low-level APDU communication, i.e., the routines that are normally responsible for sending and receiving data from the card reader. Our implementation simply assumes that communication happens behind the scenes implicitly. This is not a real limitation. During formal verification of applet code it is sufficient to specify what the contents of the APDU buffer is. Knowing that it has in fact been transported to or from the card terminal is usually not necessary.

The third gap in the implementation are the routines related to RMI dispatching. Again, this would be possible, but very difficult to implement, resulting in a unjustifiably large code. On the other hand it is very easy to verify RMI based applet code without knowing the details of how RMI methods are dispatched. That is, it is not necessary to know how a given RMI method is marshaled or unmarshaled to verify its code. Moreover, even if we did implement the RMI dispatching routines in our API, it would not be possible to reason about them with the KeY system. Such reasoning would require (at least partial) support for class and method reflection which is not present in KeY at the moment.

Finally, the Java Card platform is capable of tracking and reporting memory consumption on the card through API methods. This is implementable only to certain extent, namely, dedicated methods for allocating transient memory can

keep track of transient memory usage. Tracking persistent memory usage is not possible. In principle, this would require hooking some Java code into the built-in `new` operator. On the other hand, it would be possible to delegate the job of tracking memory usage to KeY, i.e., in principle memory usage properties could be verified during symbolic execution. The support for reasoning about memory usage properties is yet another feature currently being integrated into KeY.

Apart from that our implementation includes all of the Java implementable features specified in the official Java Card documentation [11]. Notably, the following items are taken into account.

The Java Card firewall mechanism enforces object access checks on two levels, the Java Card VM level and the API level. All checks required on the API level are included in our implementation. The routines to provide shareable interface objects to client applets across the firewall are also implemented. In the KeY system, the modelling of the checks on the VM level requires changes to the Java Card DL. The work to incorporate the firewall mechanism formalisation into Java Card DL is underway. Without this formalisation, the API firewall checks are transparent during verification. All objects in verified Java Card programs are treated as if they are owned by the Java Card system, i.e., all objects are privileged and access is always allowed.

All features related to transaction mechanism and memory types (persistent or transient) are included. In particular, it means that (i) methods specified in the documentation to be atomic utilise the transaction mechanism in a suitable way, (ii) data that is required to be transient is kept in transient memory blocks, and (iii) updates to all data that are to be excluded from the transaction mechanism are implemented in a suitable way. A notable example of the last is the PIN try counter [4]. The KeY system fully supports the Java Card transaction mechanism and different memory types, and thus all of the code involving transactions can be faithfully specified and verified with KeY.

All cryptographic interfaces (ciphers and keys) have associated implementing classes, but do not include the actual cryptographic logic as described above. A possibility to declare a cipher to be shareable or non-shareable between different applets, or for a key to implement internal key data encryption (the `KeyEncryption` interface) are both included in the implementation.

A lightweight applet registry is implemented to track applet identifiers (AID registry) and applet installation, activation, and selection. A possibility of an applet to be multi-selectable is also taken into account. The registry is minimal in the sense that it is just sufficient to provide meaningful results to methods of the API that require the applet registry functionality, e.g., the method `getAID` of the `JCSystem` class.

In the remainder of this section we give two samples of our implementation and associated specifications. The first example is the implementation of the method `partialEquals` of the `AID` class. The method is simply responsible for

comparing `length` bytes of the provided byte array to the AID bytes stored in the object. The comparison itself is simply a call to the `arrayCompare` utility method. First, however, some checks for the firewall mechanism have to be performed:

```
public final boolean partialEquals(byte[] bArray,
  short offset, byte length) throws SecurityException,
    ArrayIndexOutOfBoundsException {
  if (bArray==null) return false; // resp. documentation
  if(length > _theAID.length) return false; // resp. documentation
  // Firewall check:
  if (KeYJCSystem.jvmGetContext(KeYJCSystem.jvmGetOwner(bArray))
      != KeYJCSystem.jvmGetContext(
           KeYJCSystem.jvmGetOwner(KeYJCSystem.previousActiveObject))
  && KeYJCSystem.jvmGetPrivs(bArray) != KeYJCSystem.P_GLOBAL_ARRAY)
     throw KeYJCSystem.se; // System owned singleton instance
  // Actual comparison:
  return Util.arrayCompare(bArray, offset, _theAID, (short)0, length)==0;
}
```

The firewall check establishes that the caller of this method (`previousActiveObject`) was privileged to access the `bArray` parameter. If not, a system owned singleton instance of `SecurityException` is thrown. The reason for storing singleton instances of all exceptions is to follow the JAVA CARD paradigm of limiting the memory consumption, and also to separate system owned exceptions from applet owned ones. The calling of the method `arrayCompare` may result in an `ArrayIndexOutOfBoundsException`, which is allowed according to the documentation of `partialEquals`. The formal specification for this method is the following:

```
\programVariables { AID aidInst; boolean result;
  byte[] bArray; short offset; byte length; }

(bArray != null -> length >= 0 & offset >= 0 &
  offset + length <= bArray.length)
& {\subst AID aid; aidInst}(\includeFile "AID_inv.key";)
-> \<{
     result = aidInst.partialEquals(bArray, offset, length)@AID;
   }\> (
    (bArray = null | length > aidInst._theAID.length -> result = FALSE)
  & (bArray != null & length <= aidInst._theAID.length ->
      (result = TRUE <-> \forall int i; ( i >= 0 & i < length ->
         aidInst._theAID[i] = bArray[offset+i])))
  & {\subst AID aid; aidInst} (\includeFile "AID_inv.key";))
\modifies {result}
```

The first part of the precondition guarantees that no `ArrayIndexOutOfBoundsException` would be thrown. The second part assumes the class invariant (for easy reuse stored in a separate file) for the execution of the method:

```
  aid._theAID != null & aid._theAID.<created> = TRUE
& aid._theAID.<transient> = JCSystem.NOT_A_TRANSIENT_OBJECT
& aid._theAID.length >= 5 & aid._theAID.length <= 16
```

The byte array storing the AID should not be `null`, should be allocated in the persistent memory, and its length should be between 5 and 16 according to the documentation.

The postcondition describes the value of the result in detail. It is true if and only if the first `length` bytes in the provided array `bArray` starting at `offset` are equal to `length` bytes stored in the `_theAID` instance attribute. Additionally the invariant for the AID class has to be reestablished after the method executes. Finally, this method does not modify any data, except for the local `result` variable.

The second example we want to present is the `throwIt` method of one of the JAVA CARD specific exception classes – `TransactionException`. Although the implementation and the specification of this and sibling methods are very simple they are quite important. Such methods are frequently used both in the rest of our API implementation as well as in many JAVA CARD applets. The specific feature of these methods is that it only provides exceptional behaviour, i.e., its sole purpose is to throw a system owned instance of a given exception:

```
public static void throwIt(short reason) throws TransactionException {
  _instance.setReason(reason);
  throw _instance; }
```

The `throwIt` method is static and its execution is guarded with a corresponding static invariant, which simply says that the static attribute storing the singleton instance of the exception (`_instance`) is not `null`. Additionally, for this particular instance the instance invariant for the exception class should be maintained, which states that the `_reason` array is properly allocated in transient memory. Reason codes of exceptions should be cleared every time the card loses power, so the variable storing the reason code needs to be allocated in a transient memory. In JAVA CARD only arrays can be allocated in transient memory. Thus, the reason code has to be stored in a `short` array of size 1 instead of a simple `short` attribute. The static and the instance invariant are part both of the method's precondition and postcondition:

```
    (\includeFile "TransactionException_static_inv.key";)
  & {\subst TransactionException exc; TransactionException._instance}
        (\includeFile "TransactionException_inv.key";)
-> \<{ #catchAll(TransactionException t) {
        TransactionException.throwIt(reason)@TransactionException;
      } }\>
  ( t = TransactionException._instance & t._reason[0] = reason
  & (\includeFile "TransactionException_static_inv.key";)
  & {\subst TransactionException exc; TransactionException._instance}
      (\includeFile "TransactionException_inv.key";))
\modifies { TransactionException._instance._reason[0] }
```

This contract describes the exceptional behaviour of the method. The `#catchAll` construct declares that the method can *possibly* throw an exception of the declared type. The value `t` representing the thrown exception can be checked in the

postcondition. A null value indicates no exception (normal behaviour), a non-null value indicates that the exception indeed occurred (exceptional behaviour). In the postcondition it is required that `t` is equal to the singleton instance of the exception, and so is not null by the assumption. Thus, this postcondition requires the method to throw the exception. Finally, the postcondition also specifies that the reason code of the thrown exception (a corresponding location is included in the `\modifies` clause) is equal to the parameter of the method.

One may argue that the specification for the method `throwIt` is over-engineered, the contract for the method is actually bigger than the code of the method itself. In fact, for most of the practical applications, a much simpler specification would suffice. However, we treat specifications like this as an exercise for the KeY system. It shows that detailed verification w.r.t. complex specifications is easily achieved.

4 Verification and Experience

All of the methods have been specified and verified with the KeY system. That includes the simplest methods that just return a value of an instance attribute, but also the most complex and elaborate methods, like the `buildKey` of the `KeyBuilder` class or all of the methods of the `Cipher` implementation. The proofs were performed in a fully modular way. Whenever a method was calling another method in the API, a corresponding contract was used to discharge the method call, i.e., the proofs were always performed by contract application, instead of in-lining the code of the called method. It turned out in the process that the approach of applying method contract is the only feasible one. For a case study like this one in-lining of method calls results in proofs of unmanageable size.

The level of automation of the proofs is satisfactory, the majority of the methods are proved fully automatically, most of the rest require minor interactions, like simple quantifier instantiations. The only really heavy spots w.r.t. user interaction are loops (10 in total). Since our proof obligations require termination, a suitable specification for each of the loops has to be provided: the loop invariant, modification set, and loop variant. For at least two of the loops the loop invariant turned out to be quite complex and far from obvious just by looking at the code, a careful analysis of the open proof goals was necessary. Finally, it was not necessary to involve external tools to support verification. The KeY system allows to employ external decision procedures to discharge first order logic formulae, e.g., the Simplify theorem prover. For this case study the KeY prover was able to discharge all proof goals on its own.

On a darker side, some of the proofs were very heavy on computing resources. It was not uncommon for the prover to use up to 1.5GB of heap space and run for over an hour to finish a proof for one method. Such performance certainly

makes the round-trip specification engineering infeasible. For this case study one possible solution to this problem is to rewrite parts of the API implementation to improve on the prover performance. It turned out, for example, that the `switch` statements sometimes cause large growth of the proof. It is our belief that rewriting those `switch` statements into highly optimised `if` statements would partly solve the problem. This matter is currently under investigation. Moreover, for some of the proofs a minor modification of the KeY's automatic rule application mechanism was necessary to prevent proof size blowup. The modification in question is more of a hack that happens to work for this case study and not yet a proper solution to the problem.

Finally, the careful analysis of the JAVA CARD documentation allowed us to identify hot spots in the specification, places where actual card implementations are likely to be incorrect due to, e.g., documentation ambiguity or unusual complexity. Indeed, we did find a bug in one of the commercially sold cards. One of the firewall rules [12, Section 6.2.8.6] is ignored resulting in granting access to a shareable object in a situation where it is forbidden by the specification.

5 Conclusions and Future Work

We presented a formally verified reference implementation of the JAVA CARD API. The level of detail of the implementation is relatively high considering that the running environment of the implementation is a symbolic execution environment, the KeY verification system. This API implementation will serve us as a framework for verifying various JAVA CARD applets in our project. All of the implementation has been formally specified and verified with KeY. We found the verification process feasible, however, we do have some reservations to the performance of the KeY system. After some clean-ups and minor fixes to the code and the specification we will make the case study available on the web.

For the future we plan to look into the following. First we want to modify the API implementation code to improve on the verification performance. Secondly, our experience will be used to rectify the issues and problems we found in the KeY system (we have already communicated the most pressing issues to the KeY development team). Next we plan to implement the firewall functionality in the KeY logic. Then it will be possible to verify the API implementation again to make sure that the implemented firewall checks are consistent. The fourth step is to rewrite all of our specifications in JML. Here the work on improving the KeY's JML interface has to be finished first. Finally, it could be worthwhile to update our implementation to the newest stable version of the JAVA CARD API 2.2.2 [13], which introduced some minor updates. At the time we started our work the version 2.2.2 was not yet official. Moreover, none of the cards on the market actually implement JAVA CARD 2.2.2, thus, for the practical purpose of verifying realistic applet code the version 2.2.1 is sufficient.

Acknowledgements This work is supported by the research program Sentinels (`http://www.sentinels.nl`). Sentinels is financed by the Technology Foundation STW, the Netherlands Organisation for Scientific Research (NWO), and the Dutch Ministry of Economic Affairs. We would also like to thank Christian Haack, Erik Poll, Jesús Ravelo, and anonymous reviewers for their helpful comments.

References

1. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNAI*. Springer, 2007.
2. Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111 of *LNCS*, pages 342–363. Springer, 2006.
3. Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.
4. Engelbert Hubbers, Wojciech Mostowski, and Erik Poll. Tearing Java Cards. In *Proceedings, e-Smart 2006, Sophia-Antipolis, France, September 20–22*, 2006.
5. Engelbert Hubbers and Erik Poll. Transactions and non-atomic API calls in Java Card: Specification ambiguity and strange implementation behaviours. Department of Computer Science NIII-R0438, Radboud University Nijmegen, 2004.
6. Bart Jacobs and Erik Poll. Java program verification at Nijmegen: Developments and perspective. In *Software Security – Theories and Systems: Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4–6, 2003. Revised Papers*, volume 3233 of *LNCS*, pages 134–153. Springer, 2003.
7. Daniel Larsson and Wojciech Mostowski. Specifying Java Card API in OCL. In Peter H. Schmitt, editor, *OCL 2.0 Workshop at UML 2003*, volume 102C of *ENTCS*, pages 3–19. Elsevier, November 2004.
8. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML: A Notation for Detailed Design*. Kluwer Academic Publishers, 1999.
9. Hans Meijer and Erik Poll. Towards a full formal specification of the Java Card API. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, e-Smart 2001, Cannes, France*, volume 2140 of *LNCS*, pages 165–178. Springer, September 2001.
10. Quang Huy Nguyen and Boutheina Chetali. Certifying native Java Card API by formal refinement. In *Smart Card Research and Advanced Applications, 7th IFIP WG 8.8/11.2 International Conference, CARDIS 2006, Tarragona, Spain, April 19–21, 2006, Proceedings*, volume 3928 of *LNCS*, pages 313–328. Springer, 2006.
11. Sun Microsystems, Inc., `http://www.sun.com`. *Java Card 2.2.1 API Specification*, October 2003.
12. Sun Microsystems, Inc., `http://www.sun.com`. *Java Card 2.2.1 Runtime Environment Specification*, October 2003.
13. Sun Microsystems, Inc., `http://www.sun.com`. *Java Card 2.2.2 API Specification*, March 2006.
14. Jos Warmer and Anneke Kleppe. *The Object Constraint Language, Second Edition: Getting Your Models Ready for MDA*. Object Technology Series. Addison-Wesley, Reading/MA, 2003.