Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 102 (2004) 3–19
www.elsevier.com/locate/entcs

Specifying JAVA CARD API in OCL

Daniel Larsson[1] Wojciech Mostowski[2]

Computing Science Department
Chalmers University of Technology
Göteborg, Sweden

Abstract

We discuss the development of an OCL specification for the JAVA CARD API. The main purpose of this specification is to support and aid the verification of JAVA CARD programs in the KeY system. The main goal of the KeY system is to integrate object oriented design and formal methods. The already existing specification written in JML (JAVA Modelling Language) has been used as a starting point for the development of the OCL specification. In this paper we report on the problems that we encountered when writing the specification and their solutions, we present the most interesting parts of the specification, we report on successful verification attempts and finally we evaluate OCL and compare it to JML in the context of JAVA CARD program specification and verification.

Keywords: OCL, JML, JAVA CARD, Formal Specification, Formal Verification, Object-Oriented Design

1 Introduction

This paper reports on the development of an OCL specification for the JAVA CARD API [19]. JAVA CARD [9] is a subset of the JAVA programming language and is used to program smart cards. The JAVA CARD API (Application Programming Interface) is a set of library classes used in JAVA CARD programs. JAVA CARD API is a much smaller version of the standard JAVA API and is specifically designed for smart card programming. The OCL specification is necessary to perform formal verification of such programs when the implementation of the API classes is not available. Even if the API implementation is

[1] Email: it3lada@ituniv.se
[2] Email: woj@cs.chalmers.se

doi:10.1016/j.entcs.2003.09.001

available, having the OCL specification helps to avoid repetitive work of proving the API implementation each time API method is used in a JavaCard program. The secondary purpose of writing the specification is to document the behaviour of the JavaCard API in a formal way. We discuss the problems we encountered when writing the specification in OCL and their solutions. We present some of the most interesting parts of the specification and report on successful verification attempts of the reference implementation of JavaCard API w.r.t. our specification. Finally, we evaluate OCL and compare it to JML in the context of this work. This paper summarises results from [11].

In the following section we give more details about the background and motivation of this work. In Section 3 we give a detailed report on the development of the specification, in Section 4 we present some interesting parts of our specification, in Section 5 we evaluate OCL in the context of the presented work and finally we conclude in Section 6.

2 Background

2.1 The KeY Project

The work presented in this paper has been carried out as part of the KeY project [1,2,10]. The main goal of the KeY project is to enhance a commercial CASE (Computer Aided Software Engineering) tool with functionality for formal specification and deductive verification and, thus, to integrate formal methods into real-world software development processes. Accordingly, the design principles for the software verification component of the KeY system are:

(i) The specification language should be usable by people who do not have years of training in formal methods. The Object Constraint Language (OCL) [14], which is incorporated into the current version of the Unified Modelling Language (UML), is the specification language of our choice.

(ii) The programs that are verified should be written in a "real" object-oriented programming language. We decided to use JavaCard. This choice is motivated by the following reasons. First of all, many JavaCard applications are subject to formal verification, because they are usually security critical (e.g. authentication) and difficult to update in case a fault is discovered. At the same time the JavaCard language is easier to handle than full Java (for example, there is no concurrency and no GUI—see Section 2.2). Also, JavaCard programs are smaller than normal Java programs and thus easier to verify.

The architecture of the KeY system is shown in Figure 1. It is built on top of a commercial CASE tool (Borland Together Control Center [7]) and extends

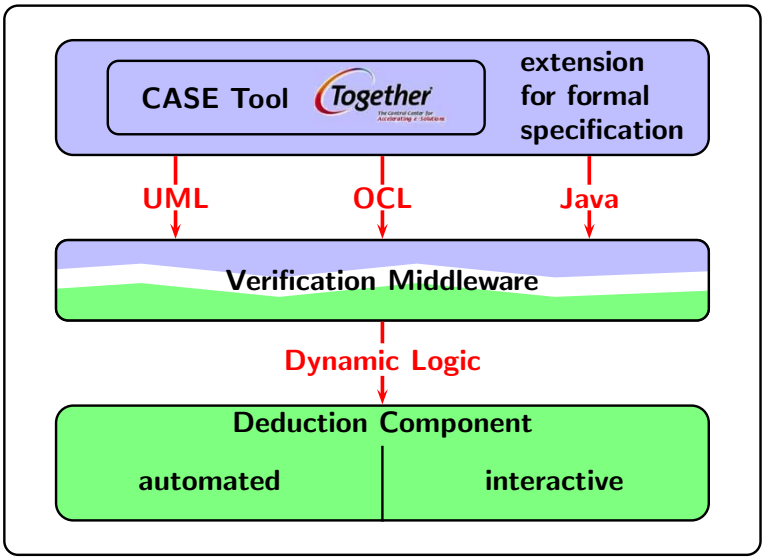

Fig. 1. The architecture of the KeY system

it with facilities for formal specification and verification of JAVA programs in the following ways:

- It supports creation and manipulation of OCL constraints, e.g. the KeY system can automatically create a partial OCL specification by instantiating an OCL template (commonly used OCL specification schema) or use a syntax based editor to create OCL expressions.
- The deduction component is used to actually construct proofs for JAVA Dynamic Logic proof obligations generated from the UML model, OCL constraints and JAVA implementation. The deduction component is an interactive verification system based on JAVA Dynamic Logic, a logic specifically designed for formal verification of JAVA programs [3].

2.2 JAVA CARD and JAVA CARD API

JAVA CARD is a technology that provides means to program smart cards with (a subset of) the JAVA programming language. Due to limited resources of smart cards, the JAVA CARD language is limited in a number of ways as compared to full JAVA. The following is the list of features that are not supported in JAVA CARD: large primitive data types (`int`, `long`, `double`, `float`), characters and strings, multidimensional arrays, dynamic class loading, threads and garbage collection. Most of the remaining JAVA features, in particular object-oriented ones like interfaces, inheritance, virtual methods, overloading, dynamic object

creation are supported by the JavaCard language.

The JavaCard API is a library that handles smart card specific features, like Application Protocol Data Units (APDUs—used for communication between the card and the rest of the world), Application IDentifiers (AIDs), JavaCard specific system routines, PIN codes, etc. [19]. Some of the packages included in the JavaCard API 2.2 are the following:

- `java.lang`—provides classes that are fundamental to the design of the JavaCard technology subset of the Java programming language. The classes in this package are derived from `java.lang` in the standard Java programming language and represent the core functionality required by the JavaCard Virtual Machine.
- `javacard.framework`—provides a framework of classes and interfaces for building, communicating and working with JavaCard applets. These classes and interfaces provide the minimum required functionality for a JavaCard environment. The key classes and interfaces in this package are:
  - `AID`—encapsulates the Application IDentifier (AID) associated with an applet.
  - `APDU`—provides methods for controlling card input and output.
  - `Applet`—the base class for all JavaCard applets on the card. It provides methods for working with applets to be loaded onto, installed into and executed on a JavaCard compliant smart card.
  - `JCSystem`—provides methods for controlling system functions such as transaction management, transient objects, object deletion mechanism, resource management, and inter-applet object sharing.
  - `Util`—provides convenience methods for working with arrays and array data.

The whole JavaCard API consists of 57 classes and interfaces, many of which are very simple (e.g. exception classes).

2.3 Use Cases for OCL Specification of the JavaCard API

One of the purposes of the KeY system is the possibility to formally verify JavaCard applications. To successfully verify a program that uses the JavaCard API one has to have access to either the implementation of the API or its formal specification. Since the implementation of the API is usually not available (especially when the methods are native), the latter is the solution we are aiming for. Let us look at an example to illustrate how the JavaCard API specification is used in the verification process. Suppose we have implemented a method `aMethod` in our JavaCard program. We now want to verify that the implementation satisfies the formal specification (the pair of

pre- and postconditions) of method `aMethod`:

```
/**
  * @preconditions <pre>
  * @postconditions <post>
  */
public void aMethod(...) {
  ...
  APIClass.apiMethod(...);
  ...
}
```

Our method invokes a method from the JavaCard API, which we assume has been already specified. The specification of `aMethod` and its implementation is translated into a proof obligation, which in turn is passed to the KeY deduction component (prover). When trying to construct a proof for this proof obligation, we sooner or later have to apply a rule that takes care of the invocation of the API method `apiMethod`. If we had no specification of this method we would have to replace the method call with the actual method body. In case the specification of `apiMethod` is available it is enough to verify that the precondition of `apiMethod` is satisfied in the state before `apiMethod` is executed and then we can simply "replace" the method call to `apiMethod` with its postcondition. This however is not as straightforward as it sounds, there is ongoing work in the KeY project which investigates when and under what conditions such a replacement can be safely done [6].

In addition to this, having an OCL specification of the API saves a lot of work during verification of JavaCard programs in the long run. When there is no specification available, the same API method call has to be replaced by the method's implementation and proved each time the method in question is used. In practice it can happen that the same piece of API implementation is going to be placed in the proof more than one time in one program.

The secondary purpose of writing the OCL specification for JavaCard API is for documentation purposes—an OCL specification can serve as formal documentation of the JavaCard API. This is very useful, because the informal specification does not always contain all the necessary information about the behaviour of the API.

2.4 Related Work

As already mentioned the starting point for this work was the formal specification of the JavaCard API written in Java Modelling Language (JML) [12,13,16]. That work has been done for similar reasons as stated above, the

main difference is the specification language used. The LOOP tool presented in [20] uses JML and PVS as the means to formally verify JavaCard programs, thus the necessity for the API specification written in JML. As we use the industry standard OCL as a specification language in the KeY project we need to have the JavaCard API specification formulated in OCL. We also made an effort to have more complete coverage of the JavaCard API in our specification.

3 The Development of OCL Specification

As stated above, we based our specification on the JML specification of the JavaCard API. We then extended it based on the informal specification (API documentation) and we tried to make use of OCL's expressiveness wherever possible. Later on we tested parts of our specification by formally verifying (using the KeY system) part of the reference implementation of the JavaCard API w.r.t. our specification.

We start by giving an overall description of JML and the JML specification of the JavaCard API. Based on that we will describe the main problems to be tackled when writing OCL specification for the API.

3.1 JML vs. OCL

As in OCL, the specifications in JML are expressed as class invariants and method pre-/postconditions. Class invariants are assertions that should hold for all instances of the class at any time. Pre- and postconditions are contracts between the provider and the user of the method. The user has to fulfil the precondition when he or she calls the method. The provider guarantees that if the precondition holds at the beginning of the method call, then the corresponding postcondition will hold after the method call. In addition, JML allows one to express when a method throws an exception and which attributes of the class can be modified by the method. All the JML specifications are only valid in the context of their Java source code and are presented in the form of Java comments. Below is the general syntax of JML used to express the method's behaviour:

```
/**
 @public behavior
   @requires <precondition>;
   @assignable <list of attributes>;
   @ensures <postcondition>;
   @signals (Exception_1 e1) <ex1postcondition>;
```

```
   @signals (Exception_2 e2) <ex2postcondition>;
 */
public void aMethod() throws Exception { ... }
```

The `@requires` clause defines the method's precondition, the `@assignable` clause tells which attributes the method can modify. The meaning of the rest of the specification is the following: if the precondition is satisfied then either the method terminates normally (i.e. does not throw any exception) and the postcondition (`@ensures`) holds or one of the listed exceptions is thrown and then the corresponding postcondition holds.

JML also allows to use a simpler syntax in case the method is not supposed to throw any exceptions, as the example below shows. The example gives a general impression of what the JavaCard API specification in JML looks like. The following is a part of the `OwnerPIN` class:

```
public class OwnerPIN implements PIN {
 private byte[] pin;
 private byte maxTries;
 private byte triesRemaining;

 public boolean check(byte[] thePin, short offset, byte length)
   throws ArrayIndexOutOfBoundsException, NullPointerException {
   ...
 }
 ...
}
```

The `pin` array contains the PIN number, `maxTries` is the maximal number of attempts allowed to present the correct PIN before the card is locked, and `triesRemaining` the number of attempts left to present the correct PIN. A JML invariant for this class is the following:

```
/**
 @invariant triesRemaining >= 0 && triesRemaining <= maxTries;
 */
```

A JML specification of the method `check` is given below. The `arrayCompare` method compares `length` elements of array `this.pin` starting at element indexed by 0 with `length` elements of array `thePin` starting at element indexed by `offset`:

```
/**
 @public normal_behavior
   @requires triesRemaining > 0 &&
```

```
   @   Util.arrayCompare(this.pin, (short)0,
   @                     thePin, offset, length) == 0;
   @ensures result == true && triesRemaining == maxTries;
*/
```

At this point we are ready to define the main differences between JML and OCL that caused us some problems when writing the JavaCard API specification in OCL. The KeY system provides extensions to OCL to overcome most of those problems.

3.2 Exceptions

The current version of OCL in its standard form does not provide a straightforward way to specify that an exception is thrown by a method. A possible solution is to have an association link `thrownExceptions` in our class, which represents the set of exceptions thrown by methods of that class. Then it is possible to specify that a method `aMethod` of class `MyClass` throws an exception of type `MyException` this way:

```
context MyClass::aMethod():
  pre:  true
  post: self.thrownExceptions->exists(e : Exception |
              e.oclIsKindOf(MyException) and e.oclIsNew())
```

The KeY system has a unified solution for this—one can use an `excThrown(MyException)` clause in the postcondition, which has a very similar meaning. Later on, when the OCL specification is transformed to a Java Dynamic Logic proof obligation for the prover, the `excThrown` clauses are properly translated to corresponding Java Dynamic Logic formulas.

Having that, we can now give the general representation of JML's `@behavior` clause in OCL:

```
context MyClass::aMethod()
pre:  <precondition>
post: (not excThrown(java::lang::Exception)
       and <postcondition>)
   or (excThrown(Exception_1) and <ex1postcondition>)
   or  ...
   or (excThrown(Exception_n) and <exnpostcondition>)
```

3.3 The `null` value

Another thing that is commonly used in Java, but which is not supported in the current version of OCL is the `null` value. This can be handled in OCL in

two ways:

- When one wants to compare a class attribute to a `null` value, then it is possible to treat the attribute as an association end, which in OCL can be treated as a set. In that case one can simple say `attr->isEmpty()` to express the fact that `attr` has a `null` value.
- When comparing objects other than class attributes (e.g. method arguments) to the `null` value things are a bit more difficult. If such an object is an array or a collection type, one can use the same technique as described above. Otherwise there is no way to specify that an object should (or should not) have the `null` value.

Fortunately, the KeY system provides a workaround for this problem as well. One can use the `null` value directly as if it were defined in OCL, and then during the translation to JAVA Dynamic Logic the `null` values are handled appropriately.

3.4 Integer Arithmetics

The main data types that JAVA CARD programs deal with are JAVA `shorts`, `bytes` and arrays. Arrays don't cause much of a problem, in OCL they can be represented as the `Sequence` type. The JAVA arithmetic types `short` and `byte` however don't have a corresponding type in OCL. The only integer type in OCL is `Integer`. The most important aspect of JAVA `shorts` and `bytes` is that they can overflow (i.e. they are finite types), while the OCL `Integer` is an infinite type and never overflows. Since the overflow behaviour is a very important aspect of JAVA programs, we have to be able to distinguish between different integer types in OCL. For this purpose we used dummy "wrapper" classes `JByte` and `JShort` to represent corresponding JAVA types. They can be used like this:

```
context PIN::check(pin: Sequence(JByte), offset: JShort,
                   length: JByte): Boolean
...
```

This still does not solve the problem of proper interpretation of overflow behaviour in OCL. Luckily, the KeY system comes to the rescue again. When the OCL specification is translated to a JAVA Dynamic Logic formula, the user can choose how the integer types are interpreted by the prover: either as finite JAVA types `short` and `byte`, or as infinite arithmetic types `arithShort` and `arithByte`. In both cases the issue of overflow is treated appropriately. More about handling arithmetics in the KeY system can be found in [5,17]. Also, [8] gives insights into problems associated with integer arithmetics in JML.

3.5 JML `assignable` clause

As mentioned before, JML offers a possibility to express (with the `@assignable` clause) that a given method is allowed to change a limited set of attributes during its execution. OCL does not offer any mechanism or language construct to specify this in a nice way. One can of course state in the post-condition that the value of a given attribute is not changed by the method by saying:

```
post: self.attr = self.attr@pre
```

This is not a good solution, though. Suppose we have a class with 20 attributes and we want to express the fact that only one attribute is assignable. That means we have to write 19 expressions like the one above for all the remaining attributes. There is ongoing work that aims at solving this problem in the KeY system [6]. The work is about how to properly specify attribute modification behaviour and how such specification can be used in proofs. In the current version of our work we left out the parts of the specification corresponding to the `@assignable` clause in JML.

4 The Specification

The present work resulted in an OCL specification for all classes and interfaces of the JavaCard API 2.2. This specification expresses, with a few exceptions (some of the `signals` clauses and the `assignable` clauses were not possible to be fully expressed in OCL), as much as the JML specification for JavaCard API 2.1.1. In some cases the OCL specification expresses more than the JML specification. In the following we illustrate by example how our OCL specification was created and how it was improved (compared to JML).

First, let us look at the `PIN` interface (which `OwnerPIN` implements). The informal specification of method `check` in the `PIN` interface is the following:

> `public boolean check(byte[] pin, short offset, byte length)`
>
> *Compares `pin` against the `PIN` value. If they match and the `PIN` is not blocked, it sets the validated flag and resets the try counter to its maximum. If it does not match, it decrements the try counter and, if the counter has reached zero, blocks the `PIN`. Even if a transaction is in progress, the internal state such as the try counter, the validated flag and the blocking state must not be conditionally updated.*
>
> ***Notes:***
> - *If `NullPointerException` or `ArrayIndexOutOfBoundsException` is thrown, the validated flag must be set to `false`, the try counter must be decremented, and the `PIN` blocked if the counter reaches zero.*
> - *If `offset` or `length` parameter is negative an `ArrayIndexOutOfBoundsException` is thrown.*
> - *If `offset+length` is greater than `pin.length`, the length of the `pin` array, an `ArrayIndexOutOfBoundsException` is thrown.*

- *If* `pin` *parameter is* `null` *a* `NullPointerException` *is thrown.*

Parameters:
`pin` *the* `byte` *array containing the* `PIN` *value being checked*
`offset` *the starting offset in the* `pin` *array*
`length` *the length of* `pin`

Returns:
`true` *if the* `PIN` *value matches;* `false` *otherwise*

Throws:
`ArrayIndexOutOfBoundsException` *if the* `check` *operation would cause access of data outside array bounds.*
`NullPointerException` *if* `pin` *is* `null`.

The JML specification for this method found in [15] is the following (the `\old` construct corresponds to OCL's `@pre`):

```
/**
 @ public normal_behavior
 @    requires triesRemaining == 0;
 @  assignable \nothing;
 @     ensures result == false;
 @ also
 @ public normal_behavior
 @    requires triesRemaining > 0 && pin != null && offset >= 0
 @             && length>=0 && offset+length == pin.length &&
 @             Util.arrayCompare(this.pin, (short)0, pin,
 @                               offset, length) == 0;
 @  assignable isValidated, triesRemaining;
 @     ensures result == true && isValidated &&
 @             triesRemaining == maxTries;
 @ also
 @ public behavior
 @    requires triesRemaining > 0 && !(pin != null &&
 @             offset >= 0 && length >= 0 &&
 @             offset+length == pin.length &&
 @             Util.arrayCompare(this.pin, (short)0, pin,
 @                               offset, length) == 0);
 @  assignable isValidated, triesRemaining;
 @     ensures result == false &&
 @             !isValidated && triesRemaining ==
 @                 \old(triesRemaining) - 1;
 @     signals (NullPointerException)
 @             !isValidated &&
```

```
 @             triesRemaining == \old(triesRemaining) - 1;
 @     signals (ArrayIndexOutOfBoundsException)
 @             !isValidated &&
 @             triesRemaining == \old(triesRemaining) - 1;
 @
 */
public boolean check(byte[] pin, short offset, byte length)
   throws ArrayIndexOutOfBoundsException, NullPointerException;
```

It seems that the JML specification mainly agrees with the informal specification. One subject that is not touched upon in the JML specification is the following sentence from the informal specification: *Even if a transaction is in progress, the internal state such as the try counter, the validated flag and the blocking state must not be conditionally updated.* This is not possible to specify in either JML or OCL, as it has to do with the internal transaction mechanism of the JavaCard Runtime Environment. The issue of specifying and verifying the programs involving JavaCard's transaction mechanism has been investigated thoroughly in the KeY project [4]. For now, however, we decided to leave this issue out in our OCL specification. Another thing to notice is that the informal specification and the JML specification disagree on the subject of whether `offset+length` must be equal to `pin.length` or if `offset+length` might be less than or equal to `pin.length`. It seems that a mistake has been made in the JML specification, since it clearly disagrees with the informal specification and since there seems to be no good reason to demand that there must be no free elements in the `pin` array following the actual PIN value. Therefore our resulting OCL specification agrees with the informal specification in this case:

```
context PIN::check(pin: Sequence(JByte), offset: JShort,
                   length: JByte): Boolean
pre:  true
post: if self.triesRemaining = 0 then result = false endif
  and if(self.triesRemaining > 0 and pin <> null
         and offset >= 0 and length >= 0 and
         offset+length <= pin->size()
         and self.pin->subSequence(1, length) =
             pin->subSequence(offset+1, offset+length))
      then (
         result = true and self.isValidated
         and self.triesRemaining = self.maxTries)
      endif
  and if(self.triesRemaining > 0 and
```

```
      not(pin <> null and offset >= 0 and length >= 0
          and offset+length <= pin->size() and
          self.pin->subSequence(1, length) =
          pin->subSequence(offset+1, offset+length)))
   then (
      not self.isValidated and
      self.triesRemaining = self.triesRemaining@pre-1 and (
        (not excThrown(java::lang::Exception) and
           result = false)
         or excThrown(NullPointerException)
         or excThrown(ArrayIndexOutOfBoundsException)))
   endif
```

In the next example we show how the specification of method `setKey` in class `DESKey` has been enriched compared to JML specification. The method `setKey` copies the data (an array of bytes) that is passed as an argument and constitutes the actual key to the internal attribute `data`. Under certain circumstances, this data is not passed to the method in plain text but as a cipher and the method must then decrypt the data before it is copied into the internal representation. Here is the JML specification for this method:

```
/**
 @public behavior
 @    requires keyData != null && kOff >= 0 &&
 @             kOff < keyData.length;
 @  assignable CryptoException.systemInstance.reason;
 @     ensures isInitialized();
 @     signals (CryptoException e)
 @       e.getReason() == CryptoException.ILLEGAL_VALUE;
 */
void setKey(byte[] keyData, short kOff) throws CryptoException;
```

This specification does not give much information about what this method actually accomplishes. In the OCL specification though, we try to give an idea about this:

```
context DESKey::setKey(keyData: Sequence(JByte), kOff: JShort)
pre:  not (keyData = null) and kOff >= 0 and
      kOff < keyData->size()
post: (not excThrown(java::lang::Exception)
       and self.isInitialized() and (
          not self.oclIsKindOf(javacardx::crypto::KeyEncryption)
       or self.getKeyCipher() = null implies
```

```
        self.data->subSequence(1, self.getSize()/8) =
        keyData->subSequence(kOff+1, kOff+self.getSize()/8)
 )
) or (
 excThrown(CryptoException) and
 CryptoException.systemInstance.reason
    = CryptoException.ILLEGAL_VALUE
 and (
   not self.oclIsKindOf(javacardx::crypto::KeyEncryption)
   or self.getKeyCipher() = null implies
        kOff+self.getSize()/8 > keyData->size()
 )
)
```

What we added in this specification is the following. If this particular instance of `DESKey` is not an instance of `javacardx.crypto.KeyEncryption` or if this instance is not associated with a `Cipher` object (the circumstances under which the input `keyData` have to be decrypted), then the input data is to be copied directly into the internal attribute `data`.

While studying the JML specification we found a small number of minor inconsistencies. In the class `OwnerPIN` for example, the invariant states that the internal class attribute `pin` should not be `null` at any point, which requires the constructor of that class to set `pin` (which is initially `null`) to a non `null` value. In that case the constructor should be able to modify the `pin` attribute, but a corresponding `@assignable` clause is missing in the specification of the `OwnerPIN` constructor. The informal specification of that constructor also says that two exceptions can be thrown—`PINException` and `SystemException`. The condition for throwing the `PINException` is clearly defined, but this information is not included in the constructor's specification.

We tried to fix all those small deficiencies in our OCL specification and express as much as possible, but, as we mentioned before, giving the full specification of the JavaCard API in OCL is not possible at the moment.

4.1 Formal Verification

To give our specification a test we looked into the source of the implementation of the JavaCard API distributed with SUN's JavaCard Development Kit version 2.1.1 [18]. We tried to verify this implementation w.r.t. the specification we have written. Due to current limitations of the KeY system this was not done to the extent one might wish for. One of the technical reasons for this is the fact that the KeY system does not handle arrays in

the version we used. Since the arrays are present almost everywhere in the JavaCard API this was a major obstacle. We can however report that a number of simple `getReason`/`setReason` methods in the exception classes of `javacard.framework` package have been verified. A more complicated successful proof attempt was the verification of the `reset` method in the `OwnerPIN` class. The specification is the following:

```
context OwnerPIN inv:
  self.maxPINSize > 0 and self.maxTries > 0 and
  self.triesRemaining >= 0 and
  self.triesRemaining <= self.maxTries

context OwnerPIN::reset()
pre:  true
post: not excThrown(java::lang::Exception)
      and
      not self.isValidated
      and
      if self.isValidated@pre then
        self.triesRemaining = self.maxTries
      else
        self.triesRemaining = self.triesRemaining@pre
      endif
```

A proof obligation generated by the KeY system states the following: the execution of the `reset` method preserves the invariant and if the precondition is satisfied before `reset` is executed then the postcondition is satisfied after `reset` is executed. Explaining what this proof obligation looks like would require introducing the Java Dynamic Logic used in the KeY system in more detail. This would go beyond the scope of this paper. One thing we should say though, is that the proof to verify this specification is performed automatically by the KeY prover, reducing the user interaction to absolute minimum.

5 Short Evaluation of OCL

There are a few things that we found very useful about OCL. First of all, it is practically an industry standard and is (partially) supported by some CASE tools (e.g. Borland Together Control Center that we use in the KeY project). Second, it seems that the OCL language is richer than JML in some respects, e.g. the whole library of collection type operations makes expressing properties about `Sequence` (array) type much easier than in JML. Also, for the same reason, we find OCL much easier to read and understand.

When it comes to Java specific features, OCL turns out to be not as good as JML. Just to recapitulate the most important findings from Section 3.1: there is no standard way in OCL to express the fact that a method throws an exception, there is only one (infinite) integer type in OCL as compared to the whole set of Java integer types and there is no JML's `@assignable` counterpart in OCL. In this respect JML is a much stronger language than OCL. Of course, this is because JML was designed specifically for Java, while OCL was mainly designed for UML.

6 Conclusions

In this paper we presented our experience from the development of an OCL specification for the JavaCard API 2.2. Despite the mentioned problems with OCL we managed to specify the whole JavaCard API to a reasonable extent. The specification is available on-line at:

http://i12www.ira.uka.de/~key/doc/2003/exjob.html

The two main purposes of this work were to aid and support formal verification of JavaCard programs in the KeY system and to document the JavaCard API in a formal way. We tested our specification by formally verifying the reference implementation of the JavaCard API with the KeY system, however, due to technical limitations, this was not done to the desirable extent. Still, the proofs we attempted were successful and were performed automatically by the KeY system. In the near future the KeY system will cover the full JavaCard standard. Then we plan to continue in this direction and also, based on our specification, perform formal verification of real life JavaCard case studies.

References

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. Technical report in computing science no. 2003-5, Department of Computing Science, Chalmers University and Göteborg University, Göteborg, Sweden, February 2003.

[2] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, and Peter H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Proceedings, Fundamental Approaches to Software Engineering (FASE), Grenoble, France*, LNCS 2306, pages 327–330. Springer, 2002.

[3] Bernhard Beckert. A dynamic logic for the formal verification of JavaCard programs. In I. Attali and T. Jensen, editors, *Revised Papers, Java on Smart Cards: Programming and Security, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[4] Bernhard Beckert and Wojciech Mostowski. A program logic for handling JavaCard's transaction mechanism. In Mauro Pezzè, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference*, LNCS 2621, pages 246–260, Warsaw, Poland, April 2003. Springer.

[5] Bernhard Beckert and Steffen Schlager. Integer arithmetic in the specification and verification of Java programs. In *Proceedings, Workshop on Tools for System Design and Verification (FM-TOOLS), Reisensburg, Germany*, pages 7–14, 2002.

[6] Bernhard Beckert and Peter H. Schmitt. Program verification using change information. In *International Conference on Software Engineering and Formal Methods, Proceedings*, September 2003. To appear.

[7] Borland Together homepage. http://www.borland.com/together/index.html.

[8] Patrice Chalin. Improving JML: For a safer and more effective language. In Stefania Gnesi, Keijiro Araki, and Dino Mandrioli, editors, *International Symposium of Formal Methods Europe, Proceedings*, LNCS 2805, pages 440–461, Pisa, Italy, September 2003. Springer.

[9] Zhiqun Chen. *Java Card Technology for Smart Cards*. Addison Wesley, 2000.

[10] KeY project homepage. http://www.key-project.org.

[11] Daniel Larsson. OCL specifications for the JavaCard API. Master's thesis, Chalmers University of Technology, Department of Computing Science, Göteborg, Sweden, 2003.

[12] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML: A Notation for Detailed Design*. Kluwer Academic Publishers, 1999.

[13] Hans Meijer and Erik Poll. Towards a full formal specification of the JavaCard API. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, e-Smart 2001, Cannes, France*. Springer-Verlag, September 2001.

[14] The Object Managment Group. *Unified Modelling Language Specification, version 1.4*, September 2001.

[15] Erik Poll. Formal interface Java specifications for the JavaCard API 2.1.1. http://www.cs.kun.nl/~erikpoll/publications/jc211_specs.html.

[16] Erik Poll, Joachim van den Berg, and Bart Jacobs. Specification of the JavaCard API in JML. CSI Report CSI-R0005, Computing Science Department, Nijmegen, March 2000.

[17] Steffen Schlager. Handling of integer arithmetic in the verification of Java programs. Diploma thesis, Department of Computer Science, Institute for Logic, Complexity and Deduction Systems, University of Karlsruhe, Germany, May 2002.

[18] Sun JavaCard developement kit 2.1.1. http://java.sun.com/products/javacard/dev_kit.html#211.

[19] Sun Microsystems, Inc. *Java Card 2.2 Application Programming Interface*, 2002. http://java.sun.com/products/javacard/specs.html.

[20] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, LNCS 2031, pages 299–312. Springer, 2001.