OCL Specifications for the Java Card API

By:

Daniel Larsson 730527-4651 GU

Supervisor:

Wojciech Mostowski

Examiner:

Wolfgang Ahrendt

Department of Computing Science
School of Computer Science and Engineering
Göteborg University

2003

Abstract

This Master's thesis discusses the development of OCL specifications for Java Card API, and is part of the KeY project. OCL is a specification language, i.e. it is used to express formally the requirements on a system. The KeY tool is a CASE tool, in which formal methods (formal specification and formal verification) are integrated with contemporary software development techniques. The main purpose of the OCL specifications is to simplify the verification of Java Card programs within the KeY tool. Verification means that one through mathematical and logical methods proves that the implementation fulfils the requirements in the specification. Already existing specifications written in JML, a specification language specially suited for Java, has been used as a starting point for the development of the OCL specifications. OCL is a more general language. Problems that have to be solved are, for instance, how to express in OCL the throwing of exceptions, how to test if a reference variable contains a null value, and how to handle the risk of overflow in the context of arithmetic integer operations. It has been shown that OCL lacks some important properties when it comes to specifying Java programs, but in other aspects is superior to JML.

Sammanfattning

Det här examensarbetet behandlar utvecklingen av OCL-specifikationer till Java Card API, och är en del av KeY-projektet. OCL är ett specifikationsspråk, d v s det används för att på ett formellt sätt uttrycka de krav man har på ett visst system. KeY är ett utvecklingsverktyg i vilket formella metoder (formell specifikation och formell verifiering) har integrerats med moderna objektorienterade utvecklingsmetoder. Syftet med OCL-specifikationerna är i första hand att underlätta verifieringen av Java Card-program i KeY. Verifiering innebär att man m h a matematiska och logiska metoder bevisar att implementeringen uppfyller kraven i specifikationen. Som utgångspunkt för OCL-specifikationerna har använts redan färdiga specifikationer skrivna i JML, ett specifikationsspråk specialanpassat till Java. OCL är ett mer generellt språk. De problem som varit tvungna att lösas är bl a hur man i OCL ska kunna uttrycka kastandet av exceptions, hur man kan testa om en referensvariabel innehåller ett null-värde och hur man hanterar risken för overflow i samband med aritmetiska heltalsoperationer. Det har visat sig att OCL saknar en del viktiga egenskaper när det gäller att specificera Java program men i andra avseenden är överlägset JML.

Preface

This report is the result of a Master's thesis in Computing Science at the Department of Computing Science, Göteborg University. Supervisor of this thesis is Wojciech Mostowski, PhD student. Wolfgang Ahrendt, assistant professor, is the examiner. They are both residing at Department of Computing Science, Göteborg University.

This work is part of the KeY project [1] - a joint project of the University of Karlsruhe and Chalmers University of Technology / Göteborg University, Gothenburg. The KeY project aims to integrate formal methods with object-oriented software development techniques

The JML specifications for the Java Card 2.1.1 API, which has been used as a starting point for this thesis, are written by Engelbert Hubbers and Erik Poll at University of Nijmegen in the Netherlands [14]. The reference implementation used [8] comes from Sun Microsystems, Inc.

Table of Contents

1. Introduction

This Master's thesis is about writing OCL [12] specifications for the Java Card API [6]. The programming language Java Card is a subset of Java, and is used to write programs for smart cards and other resource constrained devices. Smart cards are cards with an integrated circuit incorporated in the credit card-sized plastic substrate. This integrated circuit contains elements used for data transmission, storage, and processing. To communicate with the outside world, a smart card is placed in or near a card acceptance device, which is connected to a computer. Smart cards are widely used for access control, banking applications, retail loyalty applications, wireless telecommunication, and so on, where data security and privacy are major concerns [6]. OCL (Object Constraint Language) is a specification language, i.e. it is used to express requirements on software systems. One might say that a specification describes *what* the system should do, while the implementation in a programming language describes *how* it is done.

This task is part of the KeY project [1], a project that aims to integrate *formal methods* with contemporary software development techniques. Formal methods include formal specification and formal verification. "Formal" here approximately stands for "mathematically precise". In the KeY tool, which is a result of the KeY project, one can - besides doing modelling in UML (Unified Modelling Language) and implementing in Java - specify a model with OCL constraints. (OCL is in fact part of the UML standard.) Furthermore, there are facilities in the KeY tool that enable verification of the implementation w.r.t. the specification, i.e. one is able to prove that the requirements on the system, in form of OCL constraints, are satisfied when the program is run.

The whole idea of the OCL specifications for Java Card API is - in the context of KeY - to substantially simplify the verification of Java Card programs within the KeY tool. Virtually any useful Java Card program - which are called applets - uses the API. If we are able to specify the API classes and to verify a reference implementation of the API w.r.t. these specifications, then we can save a lot of time and effort at the verification of applets. We do not have to verify the API methods over and over again. They also serve as a useful documentation of the API, as they are in many aspects more clear than the informal specification. In other words, these specifications are of great interest for developers of Java Card programs.

The main purpose of this thesis is therefore to produce these useful specifications. Another purpose is - when trying to write these specifications - to evaluate the suitability of OCL as the specification language used in the KeY tool.

There do already exist specifications [14] of the Java Card API, written in JML (Java Modelling Language) [9, 10] - a specification language specially designed to specify Java programs. Therefore, the JML specifications will be used as a starting point when developing the OCL specifications. One problem, however, is that OCL has different characteristics than JML. For example, null values and the throwing of exceptions cannot be expressed in a straightforward way in OCL. Furthermore, arithmetic becomes a problem, since the built-in integer types in Java Card - `byte` and `short` - are finite, i.e. one can get an overflow when applying arithmetic operations to them, while the OCL type `Integer` is infinite. This thesis will therefore also contribute with an useful comparison between OCL and JML.

Another part of the task is to test the specifications by using a reference implementation for parts of the API [8], and then verify this implementation w. r. t. the specification, within the KeY tool.

The structure of this report is as follows. Chapter 2 contains an analysis of the problem, which leads to a method to solve it. This analysis starts with a description of the subject for-

mal methods, and how the KeY project fits in this context. Next is a short description of Java Card, and especially its API. The major part of the analysis is then concerned with an introduction of, and a comparison between, JML and OCL. How OCL and JML treat null values, exceptions, integer arithmetic and other things is considered. The analysis ends with a description of the approach used in this thesis to create the specifications. Chapter 3 contains the results of the thesis. The strengths and weaknesses of the OCL specifications - both as they are and compared to the JML specifications - are described. Here is also a description of the pro and cons of OCL as a specification language for Java. Finally, some conclusions are drawn. A selection of the produced specifications is found in appendix. All specifications produced in this thesis are available at:

http://www.mdstud.chalmers.se/~md0dala/exjob.html

2. Analysis

2.1. Formal methods and KeY

What are formal methods and what are they good for? Formal methods are techniques that are mathematically precise and are applied to improve quality or increase productivity of developments of systems that perform computations in some sense. These techniques include formal formulations of requirements, formal descriptions of constructions, and formal proofs of properties of the construction and the requirements. The *formal specification* gives an exact description of the requirements of the construction. It is the starting point for use of formal methods. Just as the formal specification constitutes a mathematical description of what the system *should do*, one also needs a mathematical description of what the system *actually does*. This description is called the *system model*. *Formal verification* means that one, through a mathematical proof, can be sure that the system (as it is described in the system model) fulfils the requirements of the system (as they are expressed in the formal specification). A proof ca be performed

- manually
- semi-automatic, with the help of computer tools
- automatic, with a computer

Automatic construction of proofs is sometimes possible, but mostly a human needs to take part in the construction by interacting with a computer tool. [16]

The KeY project [1] is about integrating formal methods with object-oriented software development. The target language of KeY-driven software development is Java. However, the verification facilities of KeY cannot handle all constructs in Java. For instance, only sequential Java can be used, i.e. not threads. Java Card, however, can be handled by the KeY tool, as it does not contain any of the “forbidden” constructs. When developing a system in the KeY tool, one may walk through the following steps:

- model the system with UML constructs
- extend this model with OCL constraints
- implement/construct the system with Java

The UML model and OCL constraints describe what the system *should do*. The Java code describes what the system *actually does*. But in this form one will not be able to verify that the implementation (Java) fulfils the requirements (UML + OCL). It has to be translated into logical formulas, and in KeY the logic used is an instance of Dynamic Logic. The result of this translation is *proof obligations*. If one is able to construct proofs of these proof obligations with the help of KeY’s interactive theorem prover, one has in fact proved that the implementation fulfils the requirements in the specification. In other words, the implementation has been verified w.r.t. the specification.

Java Card is a good target for the application of formal methods, for a number of reasons:

- Applications that run on smart cards and similar devices - and therefore can be written in Java Card - are often safety critical, security critical (e.g. access control, electronic banking), cost critical (e.g. if they run on a large number of non-administrated devices, such as phone cards), and legally critical (e.g. falling under digital signature laws). This means that the extra time and effort that the application of formal methods in these cases will lead to are highly motivated. It is extremely important that the applications behave like they should.

- The language Java Card is relatively simple, with a relatively small API, which makes the application of formal methods to it manageable.

Well, how does one specify a program? Since we are talking about object-oriented development, the units that should be specified are *classes*. A class consists of fields and methods, and when objects of this class are created, the values of the fields constitute the *state* of the object, and the methods constitute the *behaviour* of the object. Because of this, one way to specify a class is to describe what states are acceptable for the objects in this class. For instance, if there is a class `Person` with an instance field `age`, one probably does not see a negative value on `age` as something reasonable. So we might want to assert, in our specification of `Person`, that `age` must not be negative. Such an assertion is called an *invariant*. A class invariant is a proposition that has to be true for all instances (objects) of the class at any time. Another way to specify a class is to describe what the methods of the class should do, e.g. how they should alter the objects state. This can be expressed in a so-called *postcondition* of the method, in which an assertion of the system state at the end of the method invocation is made. But often one needs to assume something about the objects state and the values of the arguments that are passed to the method, to be able to make such an assertion. This assumption can be expressed in a *precondition* of the method. So the meaning of a method specification is that if the precondition is true at the beginning of the method invocation, then the postcondition will be true at the end of the method invocation. OCL supports precisely these two ways of specification - invariants and pre-/postconditions.

2.2. Java Card

Java Card provides means of programming smart cards with (a subset of) the Java programming language. Smart cards communicate with the rest of the world through application protocol data units (APDUs, ISO 7816-4 standard). The communication is done in master-slave mode - it is always the master/terminal application that initialises the communication by sending the command APDU to the card and then the card replies by sending a response APDU (possibly with empty contents). In case of Java powered smart cards (Java Cards) besides the operating system the card's ROM contains a Java Card virtual machine that implements a subset of the Java programming language and allows running Java Card applets on the card. The following are the features not supported by the Java Card language compared to full Java: large primitive data types (`int`, `long`, `double`, `float`), characters and strings, multidimensional arrays, dynamic class loading, threads and garbage collection. Some of the actual Java Card devices go beyond those limitations and support for example the `int` data type 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 Java Card language. The card also contains the standard Java Card API, which provides support for handling APDUs, Application IDentifiers (AIDs), Java Card specific system routines, PIN codes, etc. [11]

Java Card 2.2 API [7] consists of the following packages:

`java.io` - The `java.io.IOException` class is included in the Java Card API to maintain a hierarchy of exceptions identical to the standard Java programming language.

`java.lang` - Provides classes that are fundamental to the design of the Java Card 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 Java Card Virtual Machine.

`java.rmi` - The `java.rmi` package defines the `Remote` interface which identifies interfaces whose methods can be invoked from card acceptance device (CAD) client applications. It also defines a `RemoteException` that can be thrown to indicate an exception occurred during execution of a remote method call.

`javacard.framework` - Provides a framework of classes and interfaces for building, communicating with and working with Java Card applets. These classes and interfaces provide the minimum required functionality for a Java Card 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 Java Card applets on the card. It provides methods for working with applets to be loaded onto, installed into and executed on a Java Card-compliant smart card.
- `CardException`, `CardRuntimeException` - provide functionality similar to `java.lang.Exception` and `java.lang.RuntimeException` in the standard Java programming language, but specialized for the card environment.
- `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 convenient methods for working with arrays and array data.

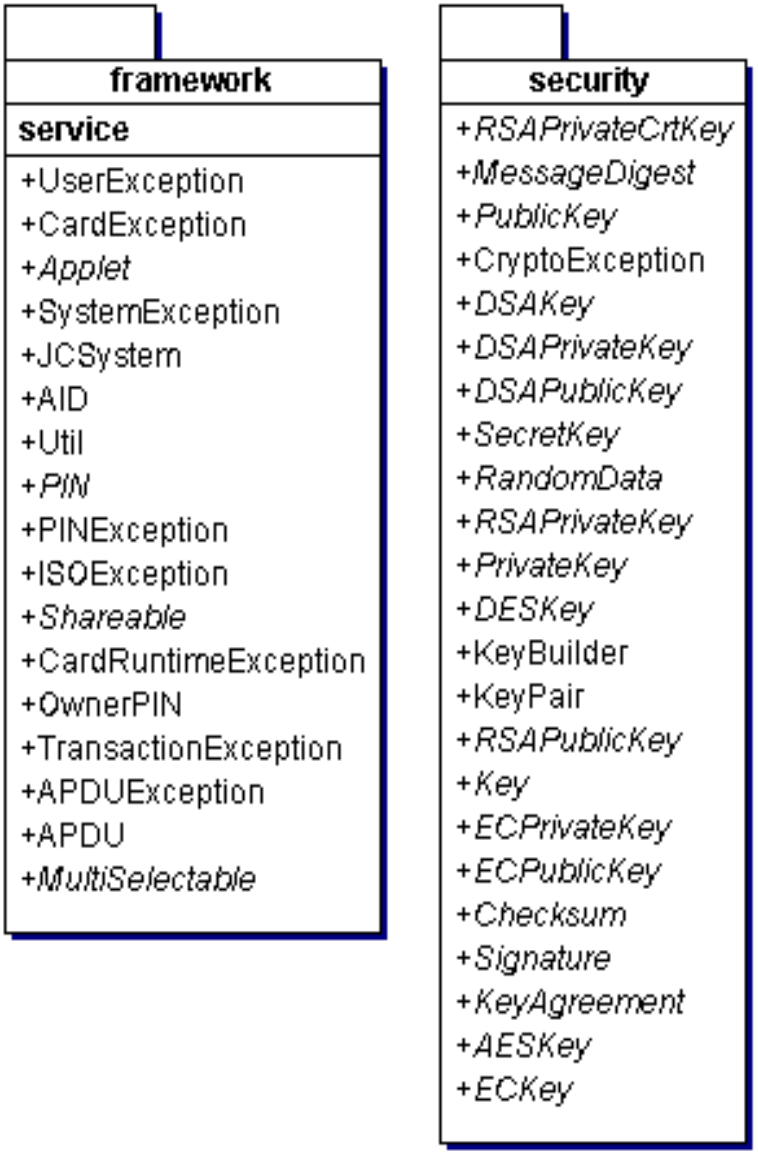
`javacard.framework.service` - Provides a service framework of classes and interfaces that allow a Java Card applet to be designed as an aggregation of service components. The package contains an aggregator class called `Dispatcher`, which includes methods to add services to its registry, dispatch APDU commands to registered services, and remove services from its registry. The package also contains the `Service` interface that contains methods to process APDU commands, and allow the dispatcher to be aware of multiple services.

`javacard.security` - Provides classes and interfaces that contain publicly-available functionality for implementing a security and cryptography framework on Java Card. Classes in the `javacard.security` package provide the definitions of algorithms that perform these security and cryptography functions:

- implementations for a variety of different cryptographic keys
- factory for building keys
- data hashing
- random data generation
- signing using cryptographic keys
- session key exchanges

`javacardx.crypto` - Extension package that contains functionality, which may be subject to export controls, for implementing a security and cryptography framework on Java Card. The package contains the `Cipher` class and the `KeyEncryption` interface. `Cipher` provides methods for encrypting and decrypting messages. `KeyEncryption` provides functionality that allows keys to be updated in a secure end-to-end fashion.

Here is an UML diagram with the packages `javacard.framework` and `javacard.security`:



2.3. Overview of JML and OCL

As has already been mentioned, the specifications used as a starting point for this project are written in JML [9, 10], while this project will result in specifications written in OCL [12]. It would therefore be rewarding to get a quick overview of these two specification languages. In both JML and OCL the most important things that can be expressed are class invariants and pre-/postconditions of methods. Class invariants are assertions that should be true for all instances of the class at any time. Pre- and postconditions can be seen as a contract between the provider and the user of the method. The user has to fulfil the precondition when he calls the method - usually by what arguments he attaches in the method call. 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.

An example of a class definition and how it could be specified with JML and OCL respectively, is the following:

```
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 {

       ...

  }

  ...
}
```

This class is actually a class in the Java Card API, and it represents a pin (personal identification number). The `pin` array contains the actual number, `maxTries` is the maximal number of tries of the user to present the correct number before the card is locked, and `triesRemaining` has the obvious meaning.

A JML invariant for this class might look like this:

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

This invariant states that the instance field `triesRemaining` must be larger than or equal to 0, and less than or equal to `maxTries`, for all objects of this class at all times.

A JML method specification of the method `check` might look like this:

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

The boolean expression in the `requires` clause is the precondition, and the expression in the `ensures` clause is the postcondition. So this clause states that if `triesRemaining` > 0 and if `thePin` that is passed to the method is equal to the real pin number, than the method will return `true` and the value of `triesRemaining` will be equal to `maxTries`[9, 10]. JML specifications actually are embedded in the source code in the form of Java comments. They cannot stand alone, as there is no way to express in JML the context in which the specification occurs. The class or method declaration is a part of the specification. OCL, on the other hand, has means to express the context of the specification and can stand alone. By using the OCL keyword `context`, one is able to express which class/interface is specified, and - in the case of pre-/postconditions - the method name, the names and types of the method parameters, and the return type of the method. The corresponding OCL invariant is therefore:

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

OCL pre- and postconditions for the method `check`, look like this:

```
context OwnerPIN::check(thePin: Sequence(JByte),
                        offset: JShort,
                        length: JByte): Boolean
  pre : self.triesRemaining > 0 and
    Util.arrayCompare(self.pin,0,thePin,offset,length) = 0
  post: result = 0 and self.triesRemaining = self.maxTries
```

OCL has in fact just one integer type - `Integer`. The use of the types `JByte` and `JShort` will be further explained later in this report, but it is a way to specify what the corresponding Java types should be in an implementation.

There is also the possibility to use OCL in the same way as JML - embedded in the source code as Java comments. That is how it is done in the Key tool. Using the OCL syntax used in the KeY tool, we would get the following specifications:

```
/**
 * @invariants
 *    self.triesRemaining >= 0 and
 *    self.triesRemaining <= self.maxTries
 */


/**
 * @preconditions
 *    self.triesRemaining > 0 and
 *    Util.arrayCompare(self.pin, 0, thePin,
 *                      offset, length) = 0
 * @postconditions
 *    result = 0 and self.triesRemaining = self.maxTries
 */
```

For more information about OCL and JML, see [12] (for OCL) and [9] (for JML).

2.4. OCL syntax used

When trying to specify Java programs with OCL, it soon becomes evident that OCL lacks properties that are necessary to produce efficient specifications in a convenient way. One is therefore forced to extend OCL with some constructs, and that is what has been done in the KeY project.

One issue is the `excThrown` construct that will be used in this report and in the resulting specifications. It is not part of the actual OCL definition, but is an extension of OCL used in the KeY tool [15]. It will be discussed later in this report. The ideal semantics of `excThrown(SomeException)` is that an instance of the class `SomeException` has been thrown in the method. This is also the semantics assumed in this project when `excThrown` is used. The implementation of `excThrown` in the current version of KeY however, can just state that some kind of exception has been thrown. One may not specify the exact type of exception.

It is also possible in KeY to check if a reference variable contains a `null` value [15], something that cannot be done in the standard OCL. This construct will also occur in the specifications, and will be touched upon later in this report.

2.5. Comparing JML and OCL

In JML there are three different constructs to specify a method; a `behavior` clause, a `normal_behavior` clause or a `exceptional_behavior` clause. The `behavior` clause is the most general one, and the other two can in fact be considered to be just syntactical sugar. A `behavior` clause looks like:

```
/*@ behavior
  @    requires <precondition>;
  @    ensures  <postcondition>;
  @    signals(Exception_1) <condition_1>;
  @    ...
  @    signals(Exception_n) <condition_n>;
  @*/
```

This 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. How could this construct be expressed in OCL? Well, a first naive try could be something like this:

```
context SomeClass::someMethod()
  pre: true
  post: if   <precondition>
        then <postcondition>
        else
           if   <condition_1>
           then excThrown(Exception_1)
           else
              if
              ...
                 if   <condition_n>
                 then excThrown(Exception_n)
                 endif
              ...
              endif
           endif
        endif
```

But this constraint does not have the same semantics as the JML clause. If one looks carefully at the definition of the behaviour clause, one sees that it states that if the precondition holds at the beginning of a method invocation, then the method either terminates normally *or terminates abruptly* ... In our naive OCL constraint we assumed that if the precondition held at the beginning of the method invocation, then the method terminates normally. Another attempt:

```
context SomeClass::someMethod()
  pre:  <precondition>
  post: <postcondition>
        or
        (excThrown(Exception_1) and <condition_1>)
        or
        ...
```

```
        or
        (excThrown(Exception_n) and <condition_n>)
```

This is definitely a better attempt. This 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. But if the method throws an exception, then we do not know that the corresponding condition will hold. Why not? It is possible that the method throws an exception, that the corresponding condition does *not* hold, but that `<postcondition>` holds. A method that behaves like that could still be verified w.r.t. this specification, which would not be the intention.
Still another attempt is necessary:

```
context SomeClass::someMethod()
  pre: <precondition>
  post:(
         not excThrown(java::lang::Exception)
         and
         <postcondition>
       )
       or
       (excThrown(Exception_1) and <condition_1>)
       or
       ...
       or
       (excThrown(Exception_n) and <condition_n>)
```

Since the class `Exception` in the package `java.lang` (is written `java::lang` in OCL syntax) is the superclass of all exceptions, this specification states the following: if the precondition holds at the beginning of a method invocation, then either the method terminates normally and `<postcondition>` holds, or otherwise one of the listed exceptions is thrown and the corresponding condition holds. Exactly what we want it to state.

Next we have the normal_behavior clause:

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

This one 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. This we can express in OCL like:

```
context SomeClass::someMethod()
  pre:  <precondition>
  post: not excThrown(java::lang::Exception)
        and
        <postcondition>
```

Finally the exceptional_behavior clause:

```
/*@ exceptional_behavior
  @    requires <precondition>;
  @    signals(SomeException)
  @*/
```

The semantics of this specification is that if the precondition holds at the beginning of a method invocation, then the method terminates abruptly by throwing a `SomeException`. The corresponding OCL constraint becomes:

```
context SomeClass::someMethod()
  pre : <precondition>
  post: excThrown(SomeException)
```

There is also the possibility in JML to put multiple clauses in one method specification, with the help of the reserved word `also`. This can be seen as a kind of case analysis. For instance it can look like this:

```
/*@ normal_behavior
  @    ...
  @ also
  @ behavior
  @    ...
  @*/
```

This simply means that both clauses must be obeyed by the method implementation, i.e. one can put a logical `and` between the clauses. So the translation to OCL is straightforward:

```
context SomeClass::someMethod()
  pre:true
  post:(
          <translation of normal_behavior clause>
       )
       and
       (
          <translation of behavior clause>
       )
```

Below is a table with a number of features in OCL and their counterparts in JML:

Comments	OCL	JML
a, b and c are boolean expressions.	not a	!a
	a and b	a && b
	a or b	a || b
	a xor b	(a || b) && !(a &&b)
	a implies b	a ==> b
	a = b	a <==> b
	if a then b else c endif	if (a) {b} else {c}
a and b are arbitrary expressions.	a = b	a == b
	a <> b	a != b

Comments	OCL	JML
a is an expression that evaluates to a variable. T is a class or interface.	a.oclIsKindOf(T)	a instanceof T
	a.oclIsNew()	\fresh(a)
	a@pre	\old(a)
When one wants to handle a collection of objects in Java Card/ JML, one usually uses an array (boolean[], byte[], short[] or Object[]). The counter-part in OCL is the Sequence type. One important difference is that Java arrays are indexed from 0 and up, while a Sequence is indexed from 1 and up.	arr->size()	arr.length
	arr->at(i)	arr[i-1]
	arr1 = arr2	arr1.equals(arr2)
	arr->subSequence(low, high)	---
	arr1->union(arr2)	---
	arr->count(obj)	---
	arr->select(expr)	---
	arr->collect(expr)	---
	arr->includes(object)	---
	arr->includesAll(collection)	---
	arr->isEmpty()	arr.length == 0
arr->iterate(elem: T; acc: T = <expr> | expression-with-elem-and-acc) The variable elem is the iterator. acc is the accu-mulator, which gets an initial value <expr>	arr->iterate(expr)	---
The general syntax of the exists and forall clauses in JML is: \exists T x; R(x) ==> P(x) where R(x) specifies the range of x.	arr->exists(x:T | P(x))	\exists T x; x>=0 && x < arr.length ==> P(x)
	arr->forAll(x:T | P(x))	\forall T x; x>=0 && x < arr.length ==> P(x)

2.6. Semantics of constraints

One subject that needs to be decided upon, before writing a specification with invariants and pre-/postconditions, is what they really mean. For instance:

- At which point in the execution of the program is the validity of an invariant enforced?
- What happens if the precondition of an operation is violated?

Let us start with the invariants. In the OCL specification of UML v1.4 we can read:
"The OCL expression can be part of an Invariant which is a Constraint stereotyped as an <<invariant>>. When the invariant is associated with a Classifier, the latter is referred to as a 'type' in this chapter. An OCL expression is an invariant of the type and must be true for all instances of that type at any time."
An interesting question is: does this mean that invariants must not even be violated during a method invocation, i.e. during intermediate computation steps? Not necessarily. The most common way to interpret an invariant seems to be that it must be true upon *completion* of the constructor and every public method. This is also how the KeY tool handles invariants.

When it comes to pre-/postcondition pairs, there are several possible semantics, which is shown in [4].
Partial Correctness - if the precondition holds at the beginning of a method invocation and if the method terminates normally, then the postcondition holds in the terminating state.

Total Correctness - if the precondition holds at the beginning of a method invocation, then the method terminates normally and the postcondition holds in the terminating state.

Partial Exception Correctness - if the precondition holds at the beginning of a method invocation and if the method terminates normally, then the postcondition holds in the terminating state; if the precondition does not hold at the beginning of a method invocation then the method does not terminate normally.

Total Exception Correctness - if the precondition holds at the beginning of a method invocation, then the method terminates normally and the postcondition holds in the terminating state; if the precondition does not hold at the beginning of a method invocation, then the method does not terminate normally.

There are still other possibilities. We have already seen the semantics of the JML `behaviour` clause:
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.

The semantics of OCL pre- and postconditions is not clear from the UML v1.4 definition, but the theorem prover in KeY however interprets a pre-/postcondition pair according to the total correctness semantics. (The subject of abruptly terminating programs is solved by the use of `excThrown`, and will be discussed later in the report.) Let us imagine we have a proof obligation as a result of a pre-/postcondition pair and an implementation that looks like:

ϕ -> <p> ψ

For deterministic programs - and Java programs are deterministic - the semantics of this proof obligation is:
For every state `s` satisfying precondition ϕ, a run of the program `p` starting in `s` terminates, and in the terminating state the postcondition ψ holds [1].

2.7. The `null` value

One problem with OCL is that there is nothing corresponding to the `null` value in Java (and JML). This is a problem that often occurs, as there is often a reason to check if a parameter reference is a `null` value, and in that case specifying the throwing of a `NullPointerException`.

Let us see how this can be handled in different situations.

```
public class MyClass {
  //@ invariant other1 != null;
  private OtherClass1 other1;

  /*@ public behavior
    @    requires <precondition>;
    @    ensures  <postcondition>;
    @    signals (java.lang.NullPointerException)
    @             other2 == null;
    @*/
  public void someMethod(OtherClass2 other2);
}
```

The invariant can be fairly accurate expressed in OCL, as an association-end can always be interpreted as an OCL `Set`. The situation above interpreted as a UML diagram would, therefore, have an association between `MyClass` and `OtherClass1`, and the association-end at `OtherClass1` would be called `other1`. If, in the context of `MyClass`, the OCL expression `self.other1` was evaluated, and the association multiplicity was `> 1`, then it would result in an OCL `Collection` (`Set`, `Bag` or `Sequence`). But even if the multiplicity is `<= 1`, then - according to the OCL definition - one has the freedom to use the result as a `Set`. This means that the invariant above, even if `other1` is a reference to just a single `Object` and not an array of `Object`s, can be expressed like this:

```
context MyClass inv:
  self.other1->notEmpty()
```

The pre- and postcondition pair is a bit harder. The parameter `other2` can be treated as a OCL `Collection` only if `OtherClass2` is of type `byte[]`, `short[]` or some kind of Java Collection type. In that case we can use the same technique as above:

```
context MyClass::someMethod(other2: OtherClass2)
  pre : <precondition>
  post: (
           not excThrown(java::lang::NullPointerException)
           and
           <postcondition>
        )
        or
        (
           excThrown(java::lang::NullPointerException)
           and
           other2->isEmpty()
         )
```

Otherwise there is no way to test for `null` in a straightforward way. However, in the KeY tool, an extension of OCL is used, that permit us to check for `null` values in a Java-like manner [15]. For instance the method specification above would then look like this:

```
context MyClass::someMethod(other2: OtherClass2)
  pre : <precondition>
  post: ...
        or
        (
```

```
            excThrown(java::lang::NullPointerException)
            and
            other2 = null
         )
```

2.8. Exceptions

There is no construct in OCL that is letting us express the throwing of an exception in a direct manner. How could we solve this? Let us assume that we in our UML diagram include the exception class `SomeException`, and that we have an association from `MyClass` to `SomeException` called `thrownExceptions`. Then we might do something like this:

```
public class MyClass {
  ...
  public SomeResultType aMethod() throws SomeException {
    if (<someCondition>)
       throw new SomeException();
    ...
  }
}


context MyClass::aMethod(): SomeResultType
  pre : true
  post: let e: SomeException
        in
              <someCondition>
           implies
              self.thrownExceptions->includes(e)
              and
              e.oclIsNew()
```

But this would not work within the KeY tool. The problem lies in how Key handles the code fragment

```
throw new SomeException();
```

The fact is that a method that terminates abruptly - e.g. if an exception is thrown and not caught - is handled as a non-terminating method. Let us see how a proof obligation is created and what it means. If we have a program (method body) `p`, a precondition ϕ and a postcondition ψ, then we will get the following proof obligation:

ϕ -> <p> ψ

This formula is valid if for every state `s` satisfying precondition ϕ, a run of the program `p` starting in `s` terminates, and in the terminating state the postcondition ψ holds. So if `p` contains a throw-statement then the KeY tool would consider it as a non-terminating program, and the proof obligation would be unsatisfiable for all postconditions ψ. This in turn means that there is no way to specify - with normal OCL constructs - that a method throws an exception, and then - within the KeY tool - verify an implementation of this method w.r.t. this specification. The way to handle this in KeY is the extra construct `excThrown` [15], which is an extension of the OCL standard and has been used earlier in this report. The idea of this construct is to get around the problem by placing the whole method body in a `try/catch-`

construct (program transformation). In the example above we would get this constraint instead:

```
context MyClass::aMethod(): SomeResultType
  pre : <<someCondition>>
  post: excThrown(SomeException)
```

And when this is translated to Dynamic Logic we would - ideally - get the proof obligation

```
==>
    <<someCondition>>
  -> <{
          boolean thrownException = false;
          try {
             self.aMethod();
          } catch(Exception thrownExc) {
             thrownException
                = thrownExc instanceof SomeException;
          }
       }> thrownException = TRUE
```

This works, because a thrown exception that is caught is not considered to result in a non-terminating program.

However, the way `excThrown` is implemented in the current version of KeY, it just states that the method does throw an `Exception`, i.e. one cannot specify what particular subclass of `Exception` that is thrown. For instance, one cannot check if the exception thrown is a `NullPointerException` or an `ArrayIndexOutOfBoundsException`. So the proof obligation above would actually look like this:

```
...
thrownException
   = thrownExc instanceof Exception;
...
```

But even if `excThrown` was implemented in the “ideal” way, it still would not have the same expressive power as the JML `signals` clause. In the `signals` clause you may use an actual instance of the thrown exception and, for instance, call the method `getReason()` defined in the `CardException` and `CardRuntimeException` classes. In other words, it works similar to the `catch` clause in the Java language. However, there is fortunately a way to get around this problem in the context of Java Card. The whole idea depends on the fact that, when programming Java Card, one is not supposed to create a new `Exception` every time one wants to throw one. Instead, `CardException`, `CardRuntimeException` and all their descendants have a static field - let us call it `systemInstance` - which holds an instance of the class itself. If a programmer wants to throw an `Exception` in his applet code, he invokes the static method `throwIt` - in the specific exception class he wants to throw - with a special reason code as argument. The `throwIt` method sets the `reason` field in the class instance held by `systemInstance`, and then throws this instance. This all has to do with the limited memory space in a smart card. Let us look at an example to clarify this:

```
public void aMethod() throws APDUException {
   ...
   APDUException.throwIt(APDUException.IO_ERROR);
}
```

The JML specification:

```
/*@
  @ behavior
  @    requires ...
  @    ensures  ...
  @    signals(APDUException e)
  @          e.getReason() == APDUException.IO_ERROR
  @*/
```

The OCL specification:

```
context SomeClass::aMethod()
  pre : ...
  post: ...
        or
        (
           excThrown(APDUException)
           and
           APDUException.systemInstance.getReason()
              = APDUException.IO_ERROR
        )
```

However, this would not work in the ordinary Java language, because then there would be no way to access the actual object that is thrown. This is a severe weakness in the `excThrown` construct, the way it is implemented today in the KeY tool.

2.9. Arithmetic

Still another difference between OCL and JML/Java is the semantics of arithmetic operations on integers. The problem is that when one applies arithmetic operations to a primitive Java integer type, `byte` or `short`, then there is a possibility that the result *overflows*. At the same time, the OCL `Integer` type behaves like a real mathematical object, i.e. it never overflows. An example will make this fact clear:

```
public byte add(byte b1, byte b2) {
  return (byte)(b1 + b2);
}
```

```
context SomeClass::add(b1: Integer, b2: Integer): Integer
  post: result = b1 + b2
```

A method invocation - `add(Byte.MAX_VALUE, 1)` - would return the value `Byte.MIN_VALUE (-128)`. At the same time the result of an evaluation of the OCL expression `b1 + b2` is `Byte.MAX_VALUE + 1 (128)`. Consequently the semantics of `byte` + and the `Integer` + is not the same. How is this solved in KeY when the implementation is to be verified w.r.t. the specification? The integers in the specification are to be treated according to the OCL semantics, when translating the specification into Dynamic Logic. But how should we handle the integers in the implementation? How are the `byte`s and `short`s treated when the implementation is translated into Dynamic Logic? That depends on what semantics we use. This issue and how it is solved in KeY is discussed in [2]. We could choose to ignore the fact that `byte`s are finite, i.e. that overflow is a possibility, and treat `byte`s as real mathematical objects. In that case the implementation above could be verified w.r.t. the specification, although the program would not fulfil the specification

under certain conditions. In other words, we might verify incorrect programs - clearly not what we want. Another alternative is to treat the `byte`s exactly as the Java Virtual Machine, i.e. to use the Java semantics. This is clearly a better choice since then we cannot verify incorrect programs. With this semantics, the implementation above cannot be verified w.r.t. the specification. This however means that the semantics of the specification integers and implementation integers are different, and this leads to other problems. For instance, formulas that are intuitively true, like - `for all x, there is a y such that y > x` - are no longer true if `x` and `y` are Java built-in types like `byte` and `short`. This problem is very obvious in the example above. Another problem is that it is easy to overlook the possiblity of overflow. This can lead to programs that are merely "incidentally" correct. This means that "a program fulfills its specification although overflow may occur, but the fact that overflow occurs was not intended neither by the modeller nor the programmer" [2]. For example, the formula

```
i > 0 -> <i=i+1; i=i-1;>(i > 0)
```

is valid although in case the value of `i` is `Byte.MAX_VALUE`, an overflow occurs and the value of `i` is (surprisingly) negative in the intermediate state after the first assignment. A more realistic example can be found in [2].

In KeY, a third alternative is used that elegantly solves the problems mentioned above. The Java syntax is extended with additional primitive data types `arithByte`, `arithShort`, `arithInt`, and `arithLong`, which are called arithmetical types in contrast to the built-in types `byte`, `short`, `int`, and `long`. In the semantics used in KeY, the additional arithmetical types basically have an infinite range. Here is a quotation from the paper [2]: "The operators acting on them have the same semantics as S_{OCL} [the semantics in which we treat the integer types as real mathematical objects] with the following restriction: If the values of the arguments of an operator are in valid range (this means, they are possible to represent in the corresponding built-in types) but the result would not (this means, overflow occurs replacing the arithmetical types with the corresponding built-in types), then the result is calculated by an invocation of the implicitly defined method `overflow(x, y, op)` whose behaviour remains unspecified. This means in case of overflow, the result is unspecified and the execution of the method `overflow` does not have to terminate. (...) If a JavaDL formula ϕ is derivable in our calculus based on S_{KeY} [the semantics used in KeY] (i.e. overflow is unspecified), then ϕ is valid in S_{KeY} for all implementations of `overflow` (this follows from the soundness of the calculus). Thus (...) one knows that no overflow occurs during the execution of `p` [`p` is the program in the formula `<p>`ϕ]." So the problem of "incidentally correct" programs can with this semantics be avoided. An implementation that uses the arithmetical types can never be verified, using this semantics, if the execution of the program may lead to overflow. If the possibility of overflow in the program is desirable, one simply uses the built-in types instead. The problem with intuitively true formulas is also solved with this approach. Arithmetical types are not allowed to occur in a program that should be compiled and executed, so - before this - they have to be replaced with the corresponding built-in types.

In the name of justice, it should be pointed out that the JML approach is not free of problems either, as shown in [5]. In JML, the semantics of integer arithmetic operations are the same as in Java, i.e. there is the possibility of overflow. Notice the difference to the situation with OCL. The problem with OCL is that the `Integer` type has an infinite range, while the Java types are finite. In JML, both the specification integer types and the implementation types have a finite range, but problems arises anyway as we shall see. Let us say we want to specify a method `intSqrt`, which should return the integer square root of its argument. A JML specification of this method could look like this:

```
/*@ public normal_behavior
  @    requires y >= 0
  @    ensures Math.abs(\result) <= y
  @            && \result * \result <= y
  @            && y < (Math.abs(\result) + 1)
  @                   * (Math.abs(\result) + 1);
  @*/
public static int intSqrt(int y)
```

Let us say that we implement this method in the following strange way:

```
public static int intSqrt(int y) {
   ...
   if (y == 0)
      return Integer.MIN_VALUE;
   ...
}
```

This implementation would be possible to verify w.r.t. the specification (if the remaining `y` values is treated in a correct way). This unexpected situation arises because operators over the `int` type in Java/JML obey the rules of modular arithmetic - thus, `Integer.MIN_VALUE = Integer.MAX_VALUE + 1`, etc. The problem is that specifiers think in terms of infinite precision arithmetic when they read and write specifications.

It would be nice to have a way to express in the OCL specification, what integer types should be used in the implementation. This is solved in this thesis by the use of the wrapper classes `JByte` and `JShort` (`J` stands for Java), which are used for specification purposes only. This means that instead of just writing

```
context AClass::aMethod(a: Integer): Integer
  pre : ...
  post: ...
```

one is able to specify the actual Java types to be used:

```
context AClass::aMethod(a: JByte): JShort
  pre : ...
  post: ...
```

The disadvantage of this is that one cannot apply the normal arithmetic operators, like +, -, *, and /, directly on objects of these types. This is due to the fact that these operators just are applicable to the OCL types `Integer` and `Real`. The solution to this problem is to have a method `asInt()` in the wrapper classes, which converts the `JByte/JShort` object to an ordinary `Integer`. An implementation of `JByte` would have an instance field called `value` or something similar. The OCL specification of the method `asInt()` would then look something like this:

```
context JByte::asInt(): Integer
  pre : true
  post: result = self.value
```

Since the calls to this method become very frequent in the specifications, and make them less clear, they have been left out in this report. That is, the `JByte` and `JShort` types are used, but we “cheat” and treat them as `Integer`s without any calls to `asInt()`. All for the sake of clarity. The method `asInt()` is however used in the original specifications. Note that the wrapper classes `JByte` and `JShort` do not solve the problems mentioned above. There is still the possibility of overflow. In the KeY tool we do not have this pro-

blem. As shown earlier in this report, the specification of a method in KeY is integrated in the source code in form of Java comments. The method declaration in Java actually becomes a part of the specification, and we therefore do not need to worry about the OCL type `Integer` when we write the specification.

2.10. The `assignable` clause in JML

There is a construct in JML, which we have not touched upon yet and that is not very easy to express in OCL - the `assignable` clause. The semantics of the `assignable` clause is that only the fields mentioned in the clause can be assigned to. Let us look at an example:

```
public class AClass {
   private byte a;
   private byte b;
   ...

   /*@ normal_behaviour
     @    requires     ...
     @    assignable   a
     @    ensures      ...
     @*/
   public void aMethod(...) {
      a = 0;
      b = 1;
   }
}
```

This implementation cannot be verified w.r.t. the specification, as `b` is not mentioned in the assignable clause but is assigned a value in the method body. There is no corresponding construct in OCL. The best one could do is to state - in the postcondition - that every field that is not mentioned in the JML assignable clause should have the same value as they did before the method invocation. The example above would look like this:

```
context AClass::aMethod(...)
  pre : ...
  post: ...
        and
        b = b@pre
```

This can, however, lead to specifications that are very hard to read. Let us, for instance, say that a class has 10 fields, and that all the methods of the class just assign a value to 1 field. This would lead to postconditions that all contain a list similar to:

```
...
and
field1 = field1@pre
and
field2 = field2@pre
and
...
and
field10 = field10@pre
```

Furthermore, a method may also assign values to non-private fields in other classes. So the list above should in fact contain all fields accessible from the context in question. To avoid these problems, the OCL specifications produced by this project do not always express the information in the assignable clauses. In cases where there seems to be a minimal risk of problems to leave it out, that has been done. There is an ongoing work to extend the OCL used in the KeY tool with a construct similar to the assignable clause in JML. More information about this can be found in [3].

2.11. Model fields

Very often in a specification one has to refer to the object state, i.e. the values of the instance fields. How do we specify an interface or a class where we do not have access to an implementation, that is, we do not have any (private) instance fields to refer to? One solution would be if we had access methods (`get`-methods) to all the relevant parts of the state, since we are allowed in our OCL constraints to use methods of the class that do not alter the object state (so-called `isQuery`-methods). But what if there are no such methods in the class, or at least not all the methods we need? (And how do we specify the get-methods?) In JML there is a way to solve this problem. One may declare so-called model fields [10], which are specification-only variables - they are used in the specification constraints to refer to the object state, but do not need to appear in an implementation of the class/interface. 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. Let us look at a real example from the JML specifications for Java Card 2.1.1 API. The specification of PIN contains a model field called `_maxTries`. It is declared like this:

```
//@public model byte _maxTries;
```

In the reference implementation of Java Card 2.1.1 API from Sun, another variable name is used to refer to this part of the object state:

```
private byte tryLimit;
```

Therefore, in order to verify the implementation w.r.t to the specification, one has to relate these variables to each other in some way. In this case it is very simple. All that is needed is an invariant like this:

```
//@ invariant _maxTries == tryLimit;
```

This means that when we refer to `_maxTries` in our specification, we also implicitly refer to `tryLimit`. Of course, the implementer of a class might want to implement the object state in a different manner than the specifier. In that case, the relations between the model fields and the variables used in the implementation would be more complicated.

Can this be expressed in OCL? Yes, this can be expressed in standard OCL, with the help of the <<definition>> constraint. This constraint must be attached to a class or interface and may only contain let definitions. The example above would look like this in OCL:

```
context PIN def:
   let _maxTries: JByte
```

And we would need a similar invariant to relate `_maxTries` to `tryLimit`:

```
context PIN inv:
   ...
   self._maxTries = self.tryLimit
```

However, this cannot be done in the current version of KeY. The KeY tool does not support the <<definition>> constraint.

2.12. Method for creating the specifications

What approach has been used when writing the OCL specifications for Java Card API? What are the guidelines that have been followed? Well, to start with, the informal specification has been read through and considered. Second, the JML specifications have been examined. Based on these sources, an effort has been made to write reasonable OCL specifications for the Java Card API. Using a formal language such as OCL, one still has to decide how detailed the specifications should be. At one end of the spectrum there are the very complete and detailed specifications, as for instance the reference implementation of the Java Card API. At the other end of the spectrum there are very incomplete or lightweight specifications that concentrate on specifying the preconditions of methods that ensure normal behaviour of the method, i.e. preconditions that rule out some - or all - unwanted run-time exceptions. Such specifications are relatively easy to write and to check, and can be used to guarantee the absence of most run-time exceptions. This is important, as omitting the proper handling of such exceptions is a common source of failures [13]. The specifications produced in this project will be somewhere in the middle. Some of the classes/interfaces will be specified in a rather lightweight manner, while others will be shown feasible to specify in more detail.

Let us look at an example that can clarify the process. In the interface `PIN`, residing in the package `javacard.framework`, there is a method `check` that checks if the pin value given by the card user agrees with the correct pin value. The informal specification [6] of this method is like follows:

```
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, 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 th rown, 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* `ArrayIndexOutOfBoundsEx ception` *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 [14] of this method looks like this:

```
/*@ 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(_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(_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;
  @*/
```

/// Some (all?) applets do not check if the `pin` & `offset` they pass on
/// to this method are ok, hence the need for the second "also behavior"
/// to say what happens if `pin==null` or the array bounds are violated.

```
public boolean check(byte[] pin, short offset, byte length)
        throws ArrayIndexOutOfBoundsException,
               NullPointerException;
```

The first `normal_behavior` clause takes care of the case when the try counter (the name `_triesRemaining` in the JML spec is definitely a better name choice) has reached zero. It states that the state cannot be altered in any way, and that the method returns `false`. The

second `normal_behavior` clause specifies what will happen if the `pin` array passed as an argument is not a `null` value, the `offset` and `length` arguments are non-negative, the `length` added to the `offset` is equal to the size of the passed `pin` array, and the pin value given by the user matches the correct pin value. It states that the method will then return `true`, the `_isValidated` field will be set to `true` and the `_triesRemaining` field will be reset, i.e. set to `_maxTries`. (The variables mentioned here starting with an underscore - _<fieldname> - are all model fields, i.e. specification-only variables.) The `behavior` clause, finally, specifies the exceptions that can be thrown and under which circumstances. A `NullPointerException` will be thrown if `pin` is a `null` reference, and an `ArrayIndexOutOfBoundsException` will be thrown if the array bounds are violated. It also states that, if an exception is thrown, then `_isValidated` will be set to `false` and `_triesRemaining` will be decremented.

It seems like 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, internal state such as the try counter, the validated flag and the blocking state must not be conditionally updated.*" This is not very easy to specify, as it has to do with atomicity aspects of the Java Card Runtime Environment, and furthermore - as has already been mentioned - the purpose of these specifications (neither the JML or the OCL specifications) is not to specify every detail of the classes. Consequently, this is not touched upon in the OCL specification either. Another thing that one might notice is the fact 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 like a mistake has been made in the JML specifications, since it clearly disagrees with the informal specification and since there seems to be no good reasons to demand that there must be no free elements in the `pin` array, following the actual pin value. The OCL specification therefore agrees with the informal specification in this case. Here is the resulting OCL specification:

```
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
```

```
      Util.arrayCompare(self.pin, 0,
                        pin, offset, length) = 0
   )
   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
         Util.arrayCompare(self.pin, 0,
                        pin, offset, length) = 0
      )
   )
   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
```

3. Results and conclusions

3.1. The specifications

This project has resulted in OCL specifications for all classes and interfaces in Java Card 2.2 API. These specifications express, with a few exceptions (the `signals` and `assignable` clauses in JML have not always been possible to express fully in OCL), as much as the JML specifications for Java Card 2.1.1 API that has been used as a starting point. With some methods, the OCL specifications express even more than the JML specifications. For instance, in the numerous interfaces and classes that extends/implements the `Key` interface in the package `javacard.security`, the specifications of the `get` and `set` methods have been somewhat extended compared to the JML specifications. Let us, as an example, look at the method `setKey` in the interface `DESKey`. This method copies the data (an array of `byte`s) that is passed as an argument and that constitutes the actual key, to the internal representation. Under certain circumstances, these data are not passed to the method in plaintext 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 [14] 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;
```

As can be seen, this specification does not give much information about what this method actually accomplishes. In the OCL specification though, there is an attempt to give an idea of 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
```

```
                Util.arrayCompare(self.data, 0,
                  keyData, kOff, self.getSize()/8) = 0
           )
        )
        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()
            )
         )
```

We see that if it is not the case that this particular instance of `DESKey` is also an instance of `javacardx.crypto.KeyEncryption` or, if it is, that this instance is not associated with a `Cipher` object (the circumstances under which the in-data have to be decrypted [6]), then the in-data is to be copied directly into the internal representation.

Let us also look at the method `getKey` in `DESKey`. This method returns the key-data in plain text. It does not, however, return it as the return value of the method, but instead one has to pass a reference to a byte array as an argument to the method. The key-data in the internal representation is then copied into this array. The return value of the method is instead the byte length of the key-data. Here is the JML specification for this method:

```
/*@ public behavior
  @    requires keyData != null && kOff >= 0 &&
  @          kOff < keyData.length && isInitialized();
  @     ensures true;
  @*/
byte getKey(byte[] keyData, short kOff);
```

We see that this specification says nothing about what the method accomplishes or what it returns. The OCL specification expresses some more:

```
context DESKey::getKey(keyData: Sequence(JByte),
                       kOff: JShort): JByte
  pre : keyData <> null
        and
        kOff >= 0
        and
        kOff < keyData->size()
        and
        self.isInitialized()
  post: result = self.getSize()/8
        and
```

```
          Util.arrayCompare(self.data, 0, keyData,
                            kOff,self.getSize()/8) = 0
```

There are a lot of similar `set` and `get` methods in the interfaces in `javacard.security`, and they are all treated in the same way as above.

Another example where the OCL specifications differ from the JML specifications is the class `OwnerPIN` in `javacard.framework`. The invariants and method specification of the constructor, in the JML specification of this class, look like this:

```
   //@ invariant 0 < _maxPINSize && 0 < _maxTries;
   //@ invariant 0 <= _triesRemaining &&
   //@              _triesRemaining <= _maxTries;
   //@ invariant (_triesRemaining == 0) <==>
   //@               (* the PIN is blocked *);
   //@ invariant _pin != null && _pin.length <= _maxPINSize;

   /*@ public normal_behavior
     @    requires maxPINSize > 0 && tryLimit > 0;
     @  assignable _maxPINSize, _maxTries, _triesRemaining,
     @             _isValidated;
     @     ensures _maxPINSize == maxPINSize &&
     @               _maxTries == tryLimit &&
     @             _triesRemaining == tryLimit &&
     @              ! _isValidated;
     @*/
public OwnerPIN(byte tryLimit, byte maxPINSize)
  throws PINException;
```

There are a number of problems here. To start with, the model field `_pin` is not in the `assignable` clause of the constructor, and at the same time there is an invariant stating that `_pin` must not be a `null`-reference at any time. These conditions are obviously not possible to satisfy. We must be allowed to assign a value to `_pin` in the constructor, otherwise `_pin` will be a `null`-reference upon completion of the constructor, and the invariant will not be satisfied. Another problem is that according to this specification, the constructor is not allowed to throw any exceptions, although the informal specification says it should be. The informal specification states that if the `maxPINSize` argument is less than 1, then the constructor should throw a `PINException` with reason code `PINException.ILLEGAL_VALUE`. Depending on how one chooses to implement this class the constructor might also throw a `SystemException` with reason code `SystemException.NO_TRANSIENT_SPACE`. This has to do with the special requirements on this class mentioned in the informal specification: Even if a transaction is in progress, internal state such as the try counter, the validated flag and the blocking state must not be conditionally updated. This leads to the OCL specification of the `OwnerPIN` constructor:

```
   context OwnerPIN::OwnerPIN(tryLimit: JByte,
                              maxPINSize: JByte)
     pre : maxPINSize > 0
           and
           tryLimit > 0
     post: (
              not excThrown(java::lang::Exception)
              and
              self.maxPINSize = maxPINSize
              and
```

```
            self.maxTries = tryLimit
            and
            self.triesRemaining = tryLimit
            and
            not self.isValidated
         )
         or
         (
             excThrown(PINException)
             and
             PINException.systemInstance.getReason()
                = PINException.ILLEGAL_VALUE
             and
             maxPINSize < 1
          )
          or
          (
             excThrown(SystemException)
             and
             SystemException.systemInstance.getReason()
                = SystemException.NO_TRANSIENT_SPACE
          )
```

A final little example. The JML specification of the `Object` class in java.lang looks like this:

```
/*@ public normal_behavior
       requires true;
    assignable \nothing;
       ensures true;
  @*/
public boolean equals(Object obj){}
```

There is a rather obvious way to improve this specification. The OCL specification looks like:

```
context Object::equals(obj: Object): Boolean
  pre : true
  post: result = (self = obj)
```

There are a number of packages (`java.io`, `java.rmi` and `javacard.framework.service`), interfaces, classes and methods that have been added in Java Card 2.2 API [7], that were not part of Java Card 2.1.1. Furthermore, the extension package `javacardx.crypto` has not been specified in the JML specification. In these cases it has not been possible to use the JML specifications as a reference, but just the informal specifications.

3.2. Verification based on the specifications

Because of the current state of the KeY tool, the specifications have not been tested to the extent one would like. There are in the KeY tool still some severe limitations that have done this impossible. For instance, the current version of KeY cannot handle methods that have

arrays as either parameter value or return value. Some method implementations have been verified w.r.t. their specification, though.

In the package `javacard.framework` there is a number of exception classes. The simple set and get methods in these classes have been verified. Let us see an example - the specification and implementation of the `CardException` methods.

```
/**
 * @invariants not (systemInstance = null)
 */
public class CardException extends Exception  {
   private static CardException systemInstance;
   private short reason;
   ...

   /**
    * @preconditions  true
    * @postconditions
         not excThrown("Exception")
         and
         result = self.reason
    */
   public short getReason() {
      return reason;
   }


   /**
    * @preconditions  true
    * @postconditions
         not excThrown("Exception")
         and
         self.reason = theReason
    */
   public void setReason(short theReason) {
      this.reason = theReason;
   }
   ...
}
```

When the KeY tool is asked to generate a proof obligation of the specification and implementation of `getReason()`, the following is the result.

```
==>
       !self.systemInstance = null
  ->  <{
           boolean thrownException = false;
           try {
              result=self.getReason();
           } catch (Exception thrownExc) {
              thrownException
                 =thrownExc instanceof Exception;
           }
        }> ( (!thrownException = TRUE
```

```
                & result = self.reason)
                 & !self.systemInstance = null)
```

The KeY tool is able to prove this proof obligation automatically, when one applies the heuristics. This means that we have proved that the implementation of `getReason()` is verified w.r.t. the specification. The proof obligation of `setReason()` is similar.

```
==>
       !self.systemInstance = null
  -> <{
          boolean thrownException = false;
          try {
             self.setReason(theReason);
          } catch (Exception thrownExc) {
             thrownException
                =thrownExc instanceof Exception;
          }
       }> ( (!thrownException = TRUE
             & self.reason = theReason)
           & !self.systemInstance = null)
```

This proof obligation is also proved automatically in KeY.

An example that is a little more complicated is the method `reset` in class `OwnerPIN`. Here is the specification and implementation (both the specification and implementation have gone through minor changes to work around the limitations in the KeY tool):

```
/**
 * @invariants
      self.maxPINSize > 0
      and
      self.maxTries > 0
      and
      self.triesRemaining >= 0
      and
      self.triesRemaining <= self.maxTries
 */
public class OwnerPIN implements PIN {
   private byte maxPINSize;
   private byte maxTries;
   private boolean isValidated;
   private byte triesRemaining;
   private byte[] pin;
   ...
   /**
    * @preconditionsnot self.isValidated
    * @postconditions
         not excThrown("Exception")
         and
         not self.isValidated
         and
         self.triesRemaining = self.triesRemaining@pre
    */
   public void reset() {
```

```
        if (isValidated())
      resetAndUnblock();
      }
   }
```

The KeY generated proof obligation looks like this:

```
   ==>
         ((((!self.isValidated = TRUE
            & self.maxPINSize > 0)
            & self.maxTries > 0)
            & self.triesRemaining >= 0)
            & self.triesRemaining <= self.maxTries)
            & all o:OwnerPIN.OwnerPIN::triesRemaining@pre(o)
               = o.triesRemaining
      -> <{
            boolean thrownException = false;
            try {
               self.reset();
            } catch(Exception thrownExc) {
               thrownException
                  = thrownExc instanceof Exception;
            }
          }> ((((((!thrownException = TRUE
                & !self.isValidated = TRUE)
                & self.triesRemaining
                     = OwnerPIN::triesRemaining@pre(self))
                & self.maxPINSize > 0)
                & self.maxTries > 0)
                & self.triesRemaining >= 0)
                & self.triesRemaining <= self.maxTries)
```

KeY is not able to construct a proof of this automatically, but simplifies the proof obligation to this:

```
      0 < self.maxPINSize,
      0 < self.maxTries,
      all o:OwnerPIN.OwnerPIN::triesRemaining@pre(o)
         = o.triesRemaining
   ==>
      self.maxTries < self.triesRemaining,
      self.triesRemaining < 0,
      self.isValidated = TRUE,
      self.triesRemaining
         = OwnerPIN::triesRemaining@pre(self)
```

After five more rule applications (manually applied) the proof obligation is proved.

3.3. The strengths of OCL

OCL is in some respects a more powerful language than JML. For instance, it is easier to compare a part of one array with a part of another array in OCL than in JML, because the built-in array method `equals` is only applicable to whole arrays. Say that we have two ar-

rays `arr1` and `arr2`. We want to check if all elements in `arr1`, from the index `off1` to index `off1+length`, are equal to the elements in `arr2`, from `off2` to `off2+length`. In JML we would have to do like this:

```
\forall short i; i >= 0 && i < length
            ==> arr1[off1+i] == arr2[off2+i]
```

In OCL it is easier to express. (Remember that the first index in a Java array is 0 while in an OCL Sequence it is 1):

```
arr1->subSequence(off1+1, off1+length)
   = arr2->subSequence(off2+1, off2+length)
```

Though it may not be particularly shorter, it is a more intuitive way to express the same thing. We can see an example of this in the specifications of `AID`. The JML specification of the constructor in `AID` looks like this:

```
/*@ public behavior
  @    requires ...
  @  assignable ...
  @     ensures length == theAID.length &&
  @              (\forall short i;  0 <= i && i < length
  @                ==> theAID [i] == bArray [offset+i]);
  @     signals (TransactionException e) ...;
  @*/
public AID( byte[] bArray, short offset, byte length )
```

The counterpart in OCL looks like this:

```
context AID::AID(bArray: Sequence(JByte),
                 offset: JShort,
                 length: JByte)
  pre : ...
  post: (
           not excThrown(java::lang::Exception)
           and
           self.theAID
            = bArray->subSequence(offset+1, offset+length)
         )
         or
         (
           excThrown(TransactionException)
           and
           ...
         )
```

Another example is OCL's own `exists` and `forAll` operations, which in many aspects are more intuitive and easier to use than their counterparts in JML. An example from the specifications of `JCSystem` will illustrate this. Here first is the JML specification of the method `makeTransientBooleanArray`:

```
 /*@ public normal_behavior
   @    requires ...
   @  assignable ...
   @     ensures (\forall byte i; 0 <= i &&
   @              i < result.length ==> result[i]==false)
```

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

Here is the OCL counterpart, which looks much nicer:

```
  context JCSystem::makeTransientBooleanArray(
             length: JShort,event: JByte): Sequence(Boolean)
    pre : ...
    post: if ...
          then
          (
             not excThrown(java::lang::Exception)
             and
             result->forAll(b: Boolean|b = false)
             and
             ...
          )
          endif
          and
          ...
```

OCL is equipped with a lot of other operations on the different `Collection` types (`Set`, `Bag` and `Sequence`) [12]. Some of these operations have already been mentioned in the table earlier in the report.

3.4. Limitations

The specifications produced in this project are limited in a number of ways. As already mentioned, they have not been tested to the desirable extent, because of “technical” problems. One also has to have in mind that some of these specifications are not trying to describe the behaviour of the methods in every detail but more concentrate on specifying the conditions of methods that ensure normal behaviour, i.e. no throwing of exceptions.

Other limitations of these specifications have to do with limitations of OCL itself. For instance, the construct `excThrown` used is not part of standard OCL, but is almost indispensable when specifying Java programs. Under certain conditions `excThrown` can however be expressed in standard OCL, but that depends on - for instance - the semantics used for abruptly terminating programs. The `null` value is also not part of standard OCL, but can be expressed to some extent with ordinary OCL constructs.

3.5. Conclusions

What are the contributions of this work?

- The specifications themselves.
  When the KeY project has made further progress, the specifications can be used within the KeY tool and substantially simplify the verification of Java Card programs. They can serve as a documentation of Java Card API that in many aspects is clearer than the infor-

mal specification from Sun. They can also work as a foundation to build on for future development towards more complete specifications for the Java Card API.

- The comparison between OCL and JML.
  The pro and cons of these languages have been made clearer. The comparison is useful if one wants to use existing JML specifications as a starting point when writing OCL specifications, and vice versa.
- Evaluation of OCL as a specification language for Java programs. Constructs that are missing in OCL when specifying Java programs has been pointed out.

Let us look at an example to illustrate the point that the specifications simplify the verification of Java Card programs. Say we have a method `aMethod` that we have specified and want to verify. This method invokes a method in the API, which has already been specified and verified.

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

This is translated into a proof obligation. When trying to construct a proof to this proof obligation, we sooner or later have to apply a rule that takes care of the invocation of the API method. If there is no specification for this method we have to replace the method call with the actual method body. But if there is a specification, and the precondition of this method is satisfied in the current state, then one may actually replace the method call with its postcondition. This means that we do not have to do the same work over and over again. The API methods are verified once and for all.

What about the evaluation of OCL as a specification language for Java programs? This work clearly shows that the OCL standard definition needs to be extended if one should be able to specify important aspects of a program written in Java or a similar object-oriented language. First and foremost, a construct that allow us to specify the throwing of exceptions in a simple but powerful way - similar to the `signals` clause in JML - needs to be added. This is very fundamental. Second, we need to be able to check if a reference variable contains a `null` value. This is also very important when specifying a Java program. There is also the problem with integer arithmetic and this might be difficult to solve in a generic way in OCL. Finally a construct similar to the `assignable` clause in JML would be very useful in OCL.

Otherwise, OCL has shown to be an expressive and elegant specification language in the context of Java Card. OCL contains a great number of operations on the `Collection` types (`Set`, `Bag` and `Sequence`), which makes it powerful and easy to use in many situations.

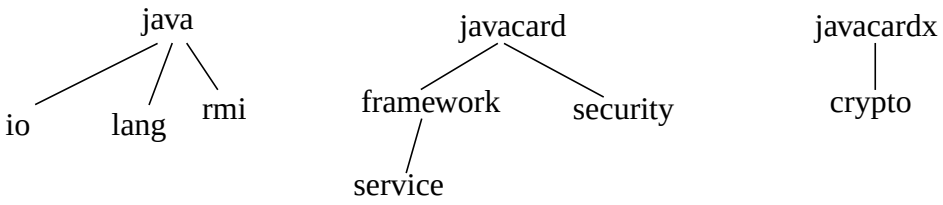
4. References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, P. H. Schmitt. *The KeY Tool*. Department of Compu ting Science, Chalmers University of Technology, Gothenburg and Department of Com puter Science, University of Karlsruhe, Karlsruhe.
2. B. Beckert, S. Schlager. *Integer Arithmetic in the Specification and Verification of Java Programs*. University of Karlsruhe, Institute for Logic, Complexity and Deduction Sys tems, Karlsruhe.
3. B. Beckert, P. H. Schmitt. *Program Verification Using Change Information*. Institute for Logic, Complexity, and Deduction Systems, Universität Karlsruhe, Germany. 2003.
4. M. Bidoit, R. Hennicker, H. Hussmann. *On the Precise Meaning of OCL Constraints*. In stitut fur Informatik, Ludvig-Maximilians-Universität Munchen, Germany and Fakultät Informatik, Technische Universität Dresden, Germany and Laboratoire Specification et Verification, CNRS & ENS de Cachan, France. 2002.
5. P. Chalin. *Back to Basics: Language Support and Semantics of Basic Infinite Integer Ty pes in JML and Larch*. Computer Science Department, Concordia University. CU-CS 2002.003.1. 2002.
6. Z. Chen. *Java Card Technology for Smart Cards*. Addison-Wesley. 2000.
7. *Java Card 2.2 Application Programming Interface*. Sun Microsystems, Inc. September, 2002.
8. *Java Card 2.1.1 Development Kit*. http://java.sun.com/products/javacard/ dev_kit.html#212. 2003-05-16
9. G. T. Leavens, A. L. Baker, C. Ruby. *JML: A Notation for Detailed Design*. Kluwer Aca demic Publishers. 1999.
10. H. Meijer, E. Poll. *Towards a full formal specification of the Java Card API*. Computing Science Institute, University of Nijmegen, The Netherlands.
11. W. Mostowski. *Towards Development of Safe and Secure Java Card Applets*. Depart- ment of Computing Science, Chalmers University of Technology/Göteborg University. 2002.
12. Object Management Group. *Unified Modelling Language Specification, version 1.4*. Sept. 2001.
13. E. Poll, J. van den Burg, B. Jacobs. *Formal specification of the JavaCard API in JML: the APDU class*. Computing Science Institute, University of Nijmegen, The Nether lands. 2001.
14. E. Poll. *Formal interface Java specifications for the Java Card API 2.1.1*. http:// www.cs.kun.nl/~erikpoll/publications/jc211_specs.html. 2003-05-16.
15. Andreas Roth. *Deduktiver Softwareentwurf am Beispiel des Java Collections Frame work - Verfeinerungsbeziehungen in UML/OCL*. Diploma thesis, University of Karlsru he, Department of Computer Science. 2002.
16. *Vad är formella metoder?*. http://www.l4i.se/S_form.htm. 2003-03-06.

Appendices

OCL specifications for the Java Card 2.2 API

Java Card 2.2 API consists of the following packages (`javacardx.crypto` is actually an extension package):



These packages are given a short description under the section 2.2, “Java Card”. The OCL specifications of these packages, which this thesis has resulted in, are available both as stand-alone OCL specifications (only OCL code) and as Java class skeletons (class and method declarations) with the specifications integrated as comments in the Java code. They can be found on the following web page:

http://www.mdstud.chalmers.se/~md0dala/exjob.html

Available on this web page is also some classes with the reference implementation given by Sun Microsystems (to Java Card 2.1.1) and OCL specifications integrated as comments. These files have been used when trying to verify some methods w.r.t. the specifications. The OCL syntax used in these files is tailored to the syntax accepted by the KeY tool.

Due to the volume of the specifications and the recurrence of similar specifications for similar methods, only a selection of the stand-alone specifications is reproduced in this appendix. Simply put, the specifications that are most interesting have been chosen - specifications that are not to trivial and that have been a challenge to specify.

`java.io` and `java.rmi` are very small and simple packages, and not very interesting in this context.

java.lang.Object

```
package java::lang


context Object::Object()
  pre : true
  post: true


context Object::equals(obj: Object): Boolean
  pre : true
  post: result = (self = obj)


endpackage
```

javacard.framework.AID

```
package javacard::framework

-- PRIVATE FIELDS
-- The variables below are not part of the informal specifica
-- tion given by SUN.
-- They are given a name, a type and a meaning that reflects a
-- part of the
-- system state, in order to be able to make a meaningful spe-
-- cification. An
-- implementer of this class is naturally free to represent -
-- the system state
-- with the help of other class and instance fields.
--
--    byte[] theAID;


context AID def:
   let theAID: Sequence(JByte)


context AID inv:
   self.theAID <> null
   and
   self.theAID->size() >= 5
   and
   self.theAID->size() <= 16

...

context AID::getBytes(dest: Sequence(JByte), offset: JShort):
JByte
  pre : true
  post: if
        (
           dest <> null
           and
           dest <> self.theAID
           -- <> is defined on OclAny. Means 'is a different
           -- object than'
           and
           offset.asInt() >= 0
           and
           offset.asInt()+self.theAID->size() <= dest->size()
        )
        then
        (
           (
```

```
            not excThrown(java::lang::Exception)
            and
            result.asInt() = self.theAID->size()
            and
           Util.arrayCompare(self.theAID, 0, dest, offset,
                                self.theAID->size()) = 0
         )
         or
         (
            excThrown(TransactionException)
            and
            TransactionException.systemInstance.
                getReason().asInt()
               = TransactionException.BUFFER_FULL
            and
            JCSystem.getTransactionDepth() = 1
         )
      )
      endif
      and
      (
         not excThrown(java::lang::Exception)
         or
         (
            excThrown(java::lang::NullPointerException)
            and
            dest = null
         )
         or
         (
            excThrown(java::lang::ArrayIndexOutOfBounds)
            and
            dest <> null
            and
            (
               offset.asInt() < 0
               or
               offset.asInt()+self.theAID->size()
                     > dest->size()
            )
         )
         or
         (
            excThrown(TransactionException)
            and
            TransactionException.systemInstance
                                    .getReason().asInt()
               = TransactionException.BUFFER_FULL
            and
            JCSystem.getTransactionDepth() = 1
```

```
            )
        )

...

endpackage
```

javacard.framework.APDU

```
package javacard::framework


-- PUBLIC FIELDS:
--    public static final byte STATE_INITIAL;
--    public static final byte STATE_PARTIAL_INCOMING;
--    public static final byte STATE_FULL_INCOMING;
--    public static final byte STATE_OUTGOING;
--    public static final byte STATE_OUTGOING_LENGTH_KNOWN;
--    public static final byte STATE_PARTIAL_OUTGOING;
--    public static final byte STATE_FULL_OUTGOING;
--    public static final byte STATE_ERROR_NO_T0_GETRESPONSE;
--    public static final byte STATE_ERROR_T1_IFD_ABORT;
--    public static final byte STATE_ERROR_IO;
--    public static final byte STATE_ERROR_NO_T0_REISSUE;
--    public static final byte PROTOCOL_MEDIA_MASK;
--    public static final byte PROTOCOL_TYPE_MASK;
--    public static final byte PROTOCOL_T0;
--    public static final byte PROTOCOL_T1;
--    public static final byte PROTOCOL_MEDIA_DEFAULT;
--    public static final byte
--           PROTOCOL_MEDIA_CONTACTLESS_TYPE_A
--    public static final byte
--           PROTOCOL_MEDIA_CONTACTLESS_TYPE_B;
--    public static final byte PROTOCOL_MEDIA_USB;
--
--
-- PRIVATE FIELDS:
-- The variables below are not part of the informal specifica
-- tion given by SUN.
-- They are given a name, a type and a meaning that reflects a
-- part of the
-- system state, in order to be able to make a meaningful spe
-- cification. An
-- implementer of this class is naturally free to represent
-- the system state
-- with the help of other class and instance fields.
--
--    //model fields in the JML spec
--    private short Lc; // incoming command length
--    private short Lr; // response length
--    private short Le; // terminal expected length
--    ////////////////////////////
--
--    private static final short BUFFERSIZE = 37;
--    private byte[] buffer;
--    private byte APDU_state;
```

```
context APDU def:
   let Lc: JShort
   let Lr: JShort
   let Le: JShort
   let BUFFERSIZE: JShort  -- static final
   let buffer: Sequence(JByte)
   let APDU_state: JByte


context APDU inv:
   self.buffer <> null
   and
   APDU.BUFFER_SIZE.asInt() >= 37
   and
   self.buffer->size() = APDU.BUFFER_SIZE.asInt()
   and
   APDU.PROTOCOL_T0 = 0
   and
   APDU.PROTOCOL_T1 = 1
   and
   self.getCurrentState().asInt() >= APDU.STATE_INITIAL
   and
   self.getCurrentState().asInt() <= APDU.STATE_FULL_OUTGOING
   and
   self.Lc.asInt() >= 0
   and
   self.Lc.asInt() < 256
   and
   self.Lr.asInt() >= 0
   and
   self.Lr.asInt() <= 256
   and
   self.Le.asInt() >= 0
   and
   self.Le.asInt() <= 256


...


context APDU::setIncomingAndReceive(): JShort
  pre : self.APDU_state = 1
        -- and
        -- 'self.Lc bytes still to be received'
  post: (
           not excThrown(java::lang::Exception)
           and
           self.APDU_state = 2
           and
           result.asInt() >= 0
           and
```

```
            result.asInt() <= self.Lc@pre
            and
            self.Lc = self.Lc@pre - result.asInt()
            and
            result.asInt()+5 <= APDU.BUFFERSIZE
            -- and
            -- 'self.Lc bytes still to be received'
            -- and
            -- 'data received in self.buffer->subSequence(6,
           --                                    6+result.asInt()-
1)'
         )
         or
         (
            excThrown(APDUException)
            and
            (
              APDUException.systemInstance.getReason().asInt()
                  = APDUException.IO_ERROR
               or
              APDUException.systemInstance.getReason().asInt()
                  = APDUException.T1_IFD_ABORT
            )
         )



context APDU::receiveBytes(bOff: JShort): JShort
  pre : self.getCurrentState().asInt()
                  = APDU.STATE_PARTIAL_INCOMING
         and
         bOff.asInt() >= 0
         and
         bOff.asInt()+self.getInBlockSize().asInt()
            <= APDU.BUFFERSIZE.asInt()
         -- and
         -- 'self.Lc.asInt() bytes still to be received'
  post: (
            not excThrown(java::lang::Exception)
            and
            (
               self.getCurrentState().asInt()
                     = APDU.STATE_PARTIAL_INCOMING
               or
               self.getCurrentState().asInt()
                        = APDU.STATE_FULL_INCOMING
            )
            and
            result.asInt() >= 0
            and
```

```
          result.asInt() <= self.Lc@pre.asInt()
          and
          self.Lc.asInt()
                   = self.Lc@pre.asInt() - result.asInt()
          and
          result.asInt()+bOff.asInt()
                   <= APDU.BUFFERSIZE.asInt()
          -- and
          -- 'self.Lc.asInt() bytes still to be received'
          -- and
          -- 'data received in self.buffer->subSequence(
        -- bOff.asInt()+1, bOff.asInt()+1+result.asInt()-1)'
       )
       or
       (
          excThrown(APDUException)
          (
            APDUException.systemInstance.getReason().asInt()
                = APDUException.IO_ERROR
             or
            APDUException.systemInstance.getReason().asInt()
                = APDUException.T1_IFD_ABORT
          )
       )



context APDU::setOutgoing(): JShort
  pre : self.getCurrentState().asInt() = APDU::STATE_INITIAL
       or
       (
          self.getCurrentState().asInt()
                  = APDU::STATE_FULL_INCOMING
          and
          self.Lc.asInt() = 0
       )
  post: (
          not excThrown(java::lang::Exception)
          and
          self.getCurrentState().asInt()
                  = APDU::STATE_OUTGOING
          and
          result.asInt() = self.Le.asInt()
          -- and
          -- 'self.Le.asInt() is the terminal expected
          -- response length'
       )
       or
       (
```

```
            excThrown(APDUException)
            and
            APDUException.systemInstance.getReason().asInt()
               = APDUException.IO_ERROR
         )



context APDU::sendBytes(bOff: JShort, len: JShort)
  pre : (
            self.getCurrentState().asInt()
               = APDU::STATE_OUTGOING_LENGTH_KNOWN
            or
            self.getCurrentState().asInt()
                   = APDU.STATE_PARTIAL_OUTGOING
         )
         and
         len.asInt() >= 0
         and
         len.asInt() <= self.Lr.asInt()
         and
         bOff.asInt()+len.asInt() <= APDU.BUFFERSIZE.asInt()
  post: (
            not excThrown(java::lang::Exception)
            and
            (
               self.getCurrentState().asInt()
                  = APDU.STATE_PARTIAL_OUTGOING
               or
               self.getCurrentState().asInt()
                         = APDU.STATE_FULL_OUTGOING
            )
            and
            self.Lr.asInt() = self.Lr@pre.asInt()-len.asInt()
            -- and
            -- (
            --       self.Lr.asInt() >= 0
            --    implies
           --       'self.buffer->subSequence(bOff.asInt()+1,
            --        bOff.asInt()+1+len.asInt()-1) sent'
            -- )
            -- and
            -- (
            --       self.Lr.asInt() = 0
            --    implies
           --       'self.buffer->subSequence(bOff.asInt()+1,
            --        bOff.asInt()+1+len.asInt()-1) will
            --        be sent later! Namely at end of current
            --                    process invocation.
            --        so self.buffer->subSequence(
```

```
            --                  bOff.asInt()+1,
          --    bOff.asInt()+1+len.asInt()-1) shouldn't
            --    be altered.'
            -- )
            -- and
            -- 'self.Lr.asInt() bytes still to be sent'
         )
         or
         (
            excThrown(APDUException)
            and
            (
              APDUException.systemInstance.getReason().asInt()
                  = APDUException.ILLEGAL_USE
               or
              APDUException.systemInstance.getReason().asInt()
                  = APDUException.IO_ERROR
               or
              APDUException.systemInstance.getReason().asInt()
                  = APDUException.NO_T0_GETRESPONSE
               or
              APDUException.systemInstance.getReason().asInt()
                  = APDUException.T1_IFD_ABORT
               or
              APDUException.systemInstance.getReason().asInt()
                  = APDUException.NO_T0_REISSUE
               or
              APDUException.systemInstance.getReason().asInt()
                  = APDUException.BUFFER_BOUNDS
            )
         )

...

endpackage
```

javacard.framework.APDUException

```
package javacard::framework


-- PRIVATE FIELDS
-- The variables below are not part of the informal specifica
-- tion given by SUN.
-- They are given a name, a type and a meaning that reflects a
-- part of the
-- system state, in order to be able to make a meaningful spe
-- cification. An
-- implementer of this class is naturally free to represent
-- the system state
-- with the help of other class and instance fields.
--
--   private static APDUException systemInstance;
--   private short reason;


context APDUException def:
   let systemInstance: APDUException  -- static
   let reason: JShort


context APDUException
   inv: APDUException.systemInstance <> null


context APDUException::APDUException(reason: JShort)
  pre : true
  post: (
          not excThrown(java::lang::Exception)
          and
          self.getReason().asInt() = reason.asInt()
        )
        or
        (
          excThrown(SystemException)
          and
          SystemException.systemInstance.getReason().asInt()
             = SystemException.NO_TRANSIENT_SPACE
        )


context APDUException::getReason(): JShort
  pre : true
  post: not excThrown(java::lang::Exception)
        and
        result.asInt() = self.reason.asInt()
```

```
context APDUException::setReason(reason: JShort)
  pre : true
  post: not excThrown(java::lang::Exception)
        and
        self.getReason().asInt() = reason.asInt()


-- static
context APDUException::throwIt(reason: JShort)
  pre : true
  post: excThrown(APDUException)
        and
        APDUException.systemInstance.getReason().asInt() =
reason.asInt()


endpackage
```

javacard.framework.Applet

```
package javacard::framework

...

-- protected final
context Applet::register(bArray: Sequence(JByte),
                         bOffset: JShort,
                         bLength: JByte)
  pre : true
  post: if
        (
           bLength.asInt() >= 5
           and
           bLength.asInt() <= 16
           and
           bOffset.asInt() >= 0
           and
           bArray->notEmpty()
           and
           bOffset.asInt()+bLength.asInt() <= bArray->size()
        )
        then
        (
           not excThrown(java::lang::Exception)
           or
           excThrown(TransactionException)
        )
        endif
        and
        (
           not excThrown(java::lang::Exception)
           or
           (
              excThrown(java::lang::NullPointerException)
              and
              bArray = null
           )
           or
           (
              excThrown(java::lang::ArrayIndexOutOfBoundsEx
ception)
              and
              (
                 bOffset.asInt() < 0
                 or
                 bLength.asInt() < 0
                 or
```

```
                bOffset.asInt()+bLength.asInt()
                            > bArray->size()
            )
         )
         or
         (
            excThrown(SystemException)
            and
            (
               (
                  SystemException.systemInstance.
           getReason().asInt()= SystemException.ILLEGAL_AID
                  and
                  (
                     bLength.asInt() < 5
                     or
                     bLength.asInt() > 16
                  )
               )
               or
               (
                  SystemException.systemInstance.
      getReason().asInt()= SystemException.ILLEGAL_VALUE
                  -- and
                  -- (
                 --  the AID bytes in bArray are already in
                  --  use
                  --  or
                 --  the RID portion of the AID bytes does
                  --  not match
                 --  the RID portion of the Java Card name
                  --  of the applet
                  --  or
                  --  a JCRE initiated install() method ex
                  --  ecution is not
                  --  in progress
                  -- )
               )
            )
         )
      )

...

endpackage
```

javacard.framework.JCSystem

```
package javacard::framework


-- PUBLIC FIELDS
--    public static final byte CLEAR_ON_DESELECT;
--    public static final byte CLEAR_ON_RESET;
--    public static final byte NOT_A_TRANSIENT_OBJECT;
--
--
-- PRIVATE FIELDS
-- The variables below are not part of the informal specifica
-- tion given by SUN.
-- They are given a name, a type and a meaning that reflects a
-- part of the
-- system state, in order to be able to make a meaningful spe
-- cification. An
-- implementer of this class is naturally free to represent
-- the system state
-- with the help of other class and instance fields.
--
-- In the context of JCSystem, there are further complica
-- tions. JCSystem does
-- not provide a piece of functionality that provides an addi
-- tion to the bare
-- JCVM and that can be understood in isolation. The parts of
-- the system state
-- that are represented by the variables below, can be changed
-- by normal Java
-- Card statements, i.e. as 'side effects' of certain virtual
-- machine instructions.
-- For instance, the variable activeContext may need to be
-- changed at every method
-- invocation. All this means that ultimately a specification
-- of the Java Card API
-- cannot be considered on its own, but has to be considered
-- together with a
-- formalisation of the Java Card language itself.
--
--    //The amount of free (i.e. unallocated) transient memory
--    private static int freeTransient;
--
--    private static byte previousContext;
--    private static byte selectedContext;
--    private static byte activeContext;
--
--    private static byte transactionDepth;
--
--    //registeredAIDs is the domain of appletTable
```

```
--    //the objects in registeredAIDs are of type AID
--    //appletTable is a partial function from AIDs to applets
--    private static Set registeredAIDs;
--    private static Map appletTable;
--
--    private static byte JCRE_CONTEXT;


context JCSystem def:
   let freeTransient: Integer
   let previousContext: JByte
   let selectedContext: JByte
   let activeContext: JByte
   let transactionDepth: JByte
   let registeredAIDs: Set
   let appletTable: (Map)
   let JCRE_CONTEXT: JByte



context JCSystem inv:
      JCSystem.NOT_A_TRANSIENT_OBJECT = 0
      and
      JCSystem.CLEAR_ON_RESET = 1
      and
      JCSystem.CLEAR_ON_DESELECT = 2
      and
      (
         JCSystem.transactionDepth.asInt() = 0
         or
         JCSystem.transactionDepth.asInt() = 1
      )

...
```

-- static

```
context JCSystem::makeTransientBooleanArray(length: JShort,
                                            event: JByte):
Sequence(Boolean)
  pre : true
  post: if
        (
           length.asInt() >= 0
           and
           length.asInt() <= JCSystem.freeTransient@pre
           and
           (
              event.asInt() = JCSystem.CLEAR_ON_RESET
              or
```

```
               event.asInt() = JCSystem.CLEAR_ON_DESELECT
            )
            and
            (
               event.asInt() = JCSystem.CLEAR_ON_DESELECT
                  implies
               JCSystem.selectedContext.asInt()
                        = JCSystem.activeContext.asInt()
            )
         )
         then
         (
            not excThrown(java::lang::Exception)
            and
            not (result = null)
            and
            result->size() = length.asInt()
            and
            result.oclIsNew()
            and
            JCSystem.isTransient(result).asInt()
                             = event.asInt()
            and
            JCSystem.freeTransient
               = JCSystem.freeTransient@pre-length.asInt()
            and
            result->forAll(b: Boolean|b = false)
         )
         endif
         and
         if
            true
         then
         (
            not excThrown(java::lang::Exception)
            or
            (
               excThrown(java::lang::NegativeArraySizeExcep
tion)
               and
               length.asInt() < 0
            )
            or
            (
               excThrown(SystemException)
               and
               (
                  (
                     SystemException.systemInstance.getRea
son().asInt()
```

```
                        = SystemException.ILLEGAL_VALUE
                     and
                     event.asInt() <> JCSystem.CLEAR_ON_RESET
                     and
                    event.asInt() <> JCSystem.CLEAR_ON_DESELECT
                  )
                  or
                  (
                     SystemException.systemInstance.getRea
son().asInt()
                        = SystemException.NO_TRANSIENT_SPACE
                     and
                     JCSystem.freeTransient < length.asInt()
                  )
                  or
                  (
                     SystemException.systemInstance.getRea
son().asInt()
                        = SystemException.ILLEGAL_TRANSIENT
                     and
                    event.asInt() = JCSystem.CLEAR_ON_DESELECT
                     and
                     JCSystem.selectedContext.asInt() <> JCSys
tem.activeContext.asInt()
                  )
               )
            )
         )
         endif

-- static
context JCSystem::beginTransaction()
  pre : true
  post: if
           JCSystem.transactionDepth.asInt@pre() = 0
         then
         (
            not excThrown(java::lang::Exception)
            and
            JCSystem.transactionDepth.asInt() = 1
         )
         endif
         and
         if
            JCSystem.transactionDepth.asInt@pre() = 1
         then
         (
            excThrown(TransactionException)
            and
            TransactionException.systemInstance.getRea
son().asInt()
```

```
              = TransactionException.IN_PROGRESS
        )
        endif



-- static
context JCSystem::abortTransaction()
  pre : true
  post: if
           JCSystem.transactionDepth.asInt@pre() = 1
        then
        (
           not excThrown(java::lang::Exception)
           and
           JCSystem.transactionDepth.asInt() = 0
        )
        endif
        and
        if
           JCSystem.transactionDepth.asInt@pre() = 0
        then
        (
           excThrown(TransactionException)
           and
           TransactionException.systemInstance.getRea
son().asInt()
              = TransactionException.NOT_IN_PROGRESS
        )
        endif



-- static
context JCSystem::commitTransaction()
  pre : true
  post: if
           JCSystem.transactionDepth.asInt@pre() = 1
        then
        (
           not excThrown(java::lang::Exception)
           and
           JCSystem.transactionDepth.asInt() = 0
        )
        endif
        and
        if
           JCSystem.transactionDepth.asInt@pre() = 0
        then
        (
           excThrown(TransactionException)
```

```
           and
           TransactionException.systemInstance.getRea
son().asInt()
              = TransactionException.NOT_IN_PROGRESS
        )
        endif


-- static
context JCSystem::getAppletShareableInterfaceObject(serve
rAID: AID,
                                          parameter: JByte):
Shareable
  pre : true
  post: if
           not (serverAID = null)
        then
           not excThrown(java::lang::Exception)
        endif
        and
        if
        (
           not (serverAID = null)
           and
           JCSystem.previousContext.asInt()
                     = JCSystem.JCRE_CONTEXT.asInt()
           and
           JCSystem.registeredAIDs.has(serverAID)
        )
        then
        (
           not excThrown(java::lang::Exception)
           and
           result =
              JCSystem.appletTable.apply(serverAID).
                 oclAsType(Applet).
                 getShareableInterfaceObject(null, parameter)
        )
        endif
        and
        if
        (
           JCSystem.previousContext.asInt() <> JCSys
tem.JCRE_CONTEXT.asInt()
           and
           JCSystem.registeredAIDs.has(serverAID)
        )
        then
        (
           not excThrown(java::lang::Exception)
           and
```

```
            result =
               JCSystem.appletTable.apply(serverAID).oclAsTy
pe(Applet).
                  getShareableInterfaceObject(
                     JCSystem.getPreviousContextAID(), parame
ter)
         )
         endif
         and
         if
         (
            not (serverAID = null)
            and
            not JCSystem.registeredAIDs.has(serverAID)
         )
         then
         (
            not excThrown(java::lang::Exception)
            and
            result = null
         )
         endif

...

endpackage
```

javacard.framework.OwnerPIN

```
package javacard::framework


-- PRIVATE FIELDS
-- The variables below are not part of the informal specifica
-- tion given by SUN.
-- They are given a name, a type and a meaning that reflects a
-- part of the
-- system state, in order to be able to make a meaningful spe
-- cification. An
-- implementer of this class is naturally free to represent
-- the system state
-- with the help of other class and instance fields.
--
--    private byte maxPINSize;
--    private byte maxTries;
--    private boolean isValidated;
--    private byte triesRemaining;
--    private byte[] pin;



context OwnerPIN def:
   let maxPINSize: JByte
   let maxTries: JByte
   let isValidated: Boolean
   let triesRemaining: JByte
   let pin: Sequence(JByte)



context OwnerPIN inv:
   self.maxPINSize.asInt() > 0
   and
   self.maxTries.asInt() > 0
   and
   self.triesRemaining.asInt() >= 0
   and
   self.triesRemaining.asInt() <= self.maxTries.asInt()
   and
   self.pin <> null
   and
   self.pin->size() <= self.maxPINSize.asInt()


...
```

```
context OwnerPIN::update(pin: Sequence(JByte),
                         offset: JShort,
                         length: JByte)
  pre : pin <> null
        and
        offset.asInt() >= 0
        and
        offset.asInt()+length.asInt() <= pin->size()
        and
        length.asInt() >= 0
  post: (
           not excThrown(java::lang::Exception)
           and
           Util.arrayCompare(self.pin, 0, pin,
                             offset, length) = 0
        )
        or
        (
           excThrown(PINException)
           and
           length.asInt() > self.maxPINSize
        )
        or
        (
           excThrown(TransactionException)
           and
           TransactionException.systemInstance.reason
              = TransactionException.BUFFER_FULL
        )


...

endpackage
```

javacard.framework.PIN

```
package javacard::framework


-- PRIVATE FIELDS
-- The variables below are not part of the informal specifica
-- tion given by SUN.
-- They are given a name, a type and a meaning that reflects a
-- part of the
-- system state, in order to be able to make a meaningful spe
-- cification. An
-- implementer of this interface is naturally free to repre
-- sent the system state
-- with the help of other class and instance fields.
--
--    private byte maxPINSize;
--    private byte maxTries;
--    private boolean isValidated;
--    private byte triesRemaining;
--    private byte[] pin;


context PIN def:
   let maxPINSize: JByte
   let maxTries: JByte
   let isValidated: Boolean
   let triesRemaining: JByte
   let pin: Sequence(JByte)


context PIN inv:
   self.maxPINSize.asInt() > 0
   and
   self.maxTries.asInt() > 0
   and
   self.triesRemaining.asInt() >= 0
   and
   self.triesRemaining.asInt() <= self.maxTries.asInt()
   and
   self.pin <> null
   and
   self.pin->size() <= self.maxPINSize.asInt()


...

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

```
  pre : true
  post: (
           if
              self.triesRemaining.asInt() = 0
           then
              result = false
           endif
        )
        and
        (
           if
           (
              self.triesRemaining.asInt() > 0
              and
              pin <> null
              and
              offset.asInt() >= 0
              and
              length.asInt() >= 0
              and
              offset.asInt()+length.asInt() <= pin->size()
              and
              Util.arrayCompare(self.pin, 0, pin, offset,
length) = 0
           )
           then
           (
              result = true
              and
              self.isValidated
              and
              self.triesRemaining.asInt() = self.max
Tries.asInt()
           )
           endif
        )
        and
        (
           if
           (
              self.triesRemaining.asInt() > 0
              and
              not
              (
                 pin <> null
                 and
                 offset.asInt() >= 0
                 and
                 length.asInt() >= 0
                 and
                 offset.asInt()+length.asInt() <= pin->size()
```

```
                and
                Util.arrayCompare(self.pin, 0, pin, offset,
length) = 0
             )
          )
          then
          (
             not self.isValidated
             and
             self.triesRemaining.asInt() = self.triesRemain
ing@pre.asInt()-1
             and
             (
                (
                   not excThrown(java::lang::Exception)
                   and
                   result = false
                )
                or
                excThrown(java::lang::NullPointerException)
                or
                excThrown(java::lang::ArrayIndexOutOf
BoundsException)
             )
          )
          endif
       )


...
endpackage
```

javacard.framework.Util

```
package javacard::framework


-- static final native
context Util::arrayCopy(src: Sequence(JByte),
      srcOff: JShort,
      dest: Sequence(JByte),
      destOff: JShort,
      length: JShort): JShort
  pre : true
  post: -- not assignable
        (
            destOff.asInt() >= 1
          implies
            dest->subSequence(1, destOff.asInt())
               = dest@pre->subSequence(1, destOff.asInt())
        )
        and
        (
            destOff.asInt()+length.asInt()+1 <= dest->size()
          implies
            dest->subSequence(de
stOff.asInt()+length.asInt()+1, dest->size())
             = dest@pre->subSequence(de
stOff.asInt()+length.asInt()+1, dest->size())
        )
        --
        and
        (
           if
           (
              src <> null
              and
              srcOff.asInt() >= 0
              and
              srcOff.asInt()+length.asInt() <= src->size()
              and
              dest <> null
              and
              destOff.asInt() >= 0
              and
              destOff.asInt()+length.asInt() <= dest->size()
              and
              length.asInt() >= 0
           )
           then
           (
              (
```

```
                  not excThrown(java::lang::Exception)
                  and
                  src@pre->subSequence(srcOff.asInt()+1, sr
cOff.asInt()+length.asInt())
                     = dest->subSequence(destOff.asInt()+1, de
stOff.asInt()+length.asInt())
               )
               or
               (
                  excThrown(TransactionException)
                  and
                  TransactionException.systemInstance.getRea
son().asInt()
                     = TransactionException.BUFFER_FULL
                  and
                  JCSystem.getTransactionDepth().asInt() = 1
               )
            )
            endif
         )
         and
         (
            if
               true
            then
            (
               not excThrown(java::lang::Exception)
               or
               (
                  excThrown(java::lang::NullPointerException)
                  and
                  (
                     src = null
                     or
                     dest = null
                  )
               )
               or
               (
                  excThrown(java::lang::ArrayIndexOutOf
BoundsException)
                  and
                  (
                     srcOff.asInt() < 0
                     or
                     destOff.asInt() < 0
                     or
                    srcOff.asInt()+length.asInt() > src->size()
                     or
                     destOff.asInt()+length.asInt() > dest->si
ze()
```

```
                  or
                  length.asInt() < 0
               )
            )
            or
            (
               excThrown(TransactionException)
               and
               TransactionException.systemInstance.getRea
son().asInt()
                  = TransactionException.BUFFER_FULL
               and
               JCSystem.getTransactionDepth().asInt() = 1
            )
         )
         endif
      )


-- static final native
context Util::arrayCompare(src: Sequence(JByte),
                           srcOff: JShort,
                           dest: Sequence(JByte),
                           destOff: JShort,
                           length: JShort): JByte
  pre : true
  post: (
         if
         (
            src <> null
            and
            srcOff.asInt() >= 0
            and
            srcOff.asInt()+length.asInt() <= src->size()
            and
            dest <> null
            and
            destOff.asInt() >= 0
            and
            destOff.asInt()+length.asInt() <= dest->size()
            and
            length.asInt() >= 0
         )
         then
         (
            not excThrown(java::lang::Exception)
            and
            (
               result.asInt() = -1
               or
               result.asInt() = 0
```

```
            or
            result.asInt() = 1
         )
         and
         (
               src->subSequence(srcOff.asInt()+1,
                         srcOff.asInt()+length.asInt())
                 = dest->subSequence(destOff.asInt()+1,
                         destOff.asInt()+length.asInt())
            implies
               result.asInt() = 0
         )
         and
         (
               Sequence{1..length.asInt()}
                           ->exists(i: Integer|
                     (
                        src->at(srcOff.asInt()+i)
                         < dest->at(destOff.asInt()+i)
                        and
                        Sequence{1..i-1}
                              ->forAll(j: Integer|
                           src->at(srcOff.asInt()+j)
                            = dest->at(destOff.asInt()+j))
                     )
            implies
               result.asInt() = -1
         )
         and
         (
               Sequence{1..length.asInt()}
                            ->exists(i: Integer|
                     (
                        src->at(srcOff.asInt()+i)
                          > dest->at(destOff.asInt()+i)
                        and
                        Sequence{1..i-1}
                                 ->forAll(j: Integer|
                           src->at(srcOff.asInt()+j)
                            = dest->at(destOff.asInt()+j))
                     )
            implies
               result.asInt() = 1
         )
      )
      endif
   )
```

```
         and
         (
            if
               true
            then
            (
               not excThrown(java::lang::Exception)
               or
               (
                  excThrown(java::lang::NullPointerException)
                  and
                  (
                     src = null
                     or
                     dest = null
                  )
               )
               or
               (
                  excThrown(java::lang::ArrayIndexOutOf
BoundsException)
                  and
                  (
                     srcOff.asInt() < 0
                     or
                     destOff.asInt() < 0
                     or
                     length.asInt() < 0
                     or
                      srcOff.asInt()+length.asInt() > src->size()
                     or
                     destOff.asInt()+length.asInt()
                            > dest->size()
                  )
               )
            )
            endif
         )

...
endpackage
```

javacard.framework.service.BasicService

```
-- import javacard.framework.*;


package javacard::framework::service


...

context BasicService::processDataIn(apdu: APDU): Boolean
  pre : apdu.getCurrentState().asInt() = APDU.STATE_INITIAL
  or
  apdu.getCurrentState().asInt() = APDU.STATE_FULL_INCOMING
  post: not excThrown(java::lang::Exception)
  and
  (
     apdu.getCurrentState().asInt() = AP
DU.STATE_FULL_INCOMING
     or
     apdu.getCurrentState().asInt() = APDU.STATE_OUTGOING
  )



context BasicService::processCommand(apdu: APDU): Boolean
  pre : apdu.getCurrentState().asInt() = APDU.STATE_INITIAL
  or
  apdu.getCurrentState().asInt() = APDU.STATE_FULL_INCOMING
  or
  apdu.getCurrentState().asInt() = APDU.STATE_OUTGOING
  post: not excThrown(java::lang::Exception)
  and
  apdu.getCurrentState().asInt() = APDU.STATE_OUTGOING


context BasicService::receiveInData(apdu: APDU): JShort
  pre : true
  post: not excThrown(java::lang::Exception)
  or
  (
     excThrown(ServiceException)
     and
     (
        (
           ServiceException.systemInstance.getRea
son().asInt()
              = ServiceException.CANNOT_ACCESS_IN_COMMAND
           and
           apdu.getCurrentState().asInt() <> AP
DU::STATE_INITIAL
```

```
            and
            apdu.getCurrentState().asInt()
                     <> APDU::STATE_FULL_INCOMING
         )
         or
         ServiceException.systemInstance.getReason().asInt()
            = ServiceException.COMMAND_DATA_TOO_LONG
      )
   )

...

endpackage
```

javacard.framework.service.Dispatcher

```
-- import java.lang.*;
-- import javacard.framework.*;


package javacard::framework::service

...

context Dispatcher::addService(service: Service, phase: JBy
te)
  pre : true
  post: not excThrown(java::lang::Exception)
  or
  (
     excThrown(ServiceException)
     and
     (
        (
           ServiceException.systemInstance.getRea
son().asInt()
              = ServiceException.ILLEGAL_PARAM
           and
           (
              (
                 phase.asInt() <> Dispatcher.PROCESS_NONE
           and
                 phase.asInt() <> Dispat
cher.PROCESS_INPUT_DATA
           and
                 phase.asInt() <> Dispatcher.PROCESS_COMMAND
           and
                 phase.asInt() <> Dispat
cher.PROCESS_OUTPUT_DATA
              )
              or
              service = null
           )
        )
        or
        ServiceException.systemInstance.getReason().asInt()
           = ServiceException.DISPATCH_TABLE_FULL
     )
        )



context Dispatcher::removeService(service: Service, phase:
JByte)
```

```
  pre : true
  post: not excThrown(java::lang::Exception)
  or
  (
           excThrown(ServiceException)
     and
     ServiceException.systemInstance.getReason().asInt()
        = ServiceException.ILLEGAL_PARAM
     and
     (
        (
           phase.asInt() <> Dispatcher.PROCESS_NONE
     and
           phase.asInt() <> Dispatcher.PROCESS_INPUT_DATA
     and
           phase.asInt() <> Dispatcher.PROCESS_COMMAND
     and
           phase.asInt() <> Dispatcher.PROCESS_OUTPUT_DATA
        )
        or
        service = null
     )
  )



context Dispatcher::dispatch(command: APDU, phase: JByte):
Exception
  pre : true
  post: not excThrown(java::lang::Exception)
  or
  (
     excThrown(ServiceException)
     and
     ServiceException.systemInstance.getReason().asInt()
        = ServiceException.ILLEGAL_PARAM
     and
     phase.asInt() <> Dispatcher.PROCESS_INPUT_DATA
           and
     phase.asInt() <> Dispatcher.PROCESS_COMMAND
           and
    phase.asInt() <> Dispatcher.PROCESS_OUTPUT_DATA
  )

...

endpackage
```

javacard.framework.service.RMIService

```
-- import java.rmi.*;


package javacard::framework::service

...

context RMIService::processCommand(apdu: APDU): Boolean
  pre : true
  post: (
           not excThrown(java::lang::Exception)
           and
           (
              apdu.getCurrentState().asInt() = AP
DU.STATE_INITIAL
              or
              apdu.getCurrentState().asInt() = AP
DU.STATE_FULL_INCOMING
           )
        )
        or
        (
           excThrown(ServiceException)
           and
           (
              (
                 ServiceException.systemInstance.getRea
son().asInt()
                  = ServiceException.CANNOT_ACCESS_IN_COMMAND
                 and
                 apdu.getCurrentState().asInt() <> AP
DU.STATE_INITIAL
                 or
                 apdu.getCurrentState().asInt() <> AP
DU.STATE_FULL_INCOMING
              )
              or
              ServiceException.systemInstance.getRea
son().asInt()
                = ServiceException.REMOTE_OBJECT_NOT_EXPORTED
           )
        )
        or
        excThrown(java::lang::SecurityException)

...

endpackage
```

javacard.framework.service.SecurityService

```
package javacard::framework::service

...

context SecurityService::isCommandSecure(properties: JByte):
Boolean
  pre : true
  post: not excThrown(java::lang::Exception)
        or
        (
           excThrown(ServiceException)
           and
          ServiceException.systemInstance.getReason().asInt()
              = ServiceException.ILLEGAL_PARAM
           and
           properties.asInt()
              <> SecurityServi
ce.PROPERTY_INPUT_CONFIDENTIALITY
           and
           properties.asInt() <> SecurityServi
ce.PROPERTY_INPUT_INTEGRITY
           and
           properties.asInt()
              <> SecurityServi
ce.PROPERTY_OUTPUT_CONFIDENTIALITY
           and
           properties.asInt() <> SecurityServi
ce.PROPERTY_OUTPUT_INTEGRITY
        )

...

endpackage
```

javacard.security.KeyBuilder

```
package javacard::security


-- static
context KeyBuilder::buildKey(keyType: JByte,
                             keyLength: JShort,
                             keyEncrypt: Boolean): Key
  pre : true
  post: not excThrown(java::lang::Exception)
        or
        (
           excThrown(CryptoException)
           and
           CryptException.systemInstance.getReason().asInt()
              = CryptoException.NO_SUCH_ALGORITHM
        )

endpackage
```

javacard.security.MessageDigest

```
package javacard::security

...

-- abstract
context MessageDigest::doFinal(inBuff: Sequence(JByte),
                               inOffset: JShort,
                               inLength: JShort,
                               outBuff: Sequence(JByte),
                               outOffset: JShort): JShort
  pre : inBuff <> null
        and
        outBuff <> null
        and
        inOffset.asInt() >= 0
        and
        inLength.asInt() >= 0
        and
        inOffset.asInt()+inLength.asInt() <= inBuff->size()
        and
        outOffset.asInt() <= outBuff->size()
  post: true


-- abstract
context MessageDigest::update(inBuff: Sequence(JByte),
                              inOffset: JShort,
                              inLength: JShort)
  pre : inBuff <> null
        and
        inOffset.asInt() >= 0
        and
        inLength.asInt() >= 0
        and
        inOffset.asInt()+inLength.asInt() <= inBuff->size()
  post: true

...

endpackage
```

javacard.security.RSAPrivateKey

```
package javacard::security


-- PRIVATE FIELDS
-- The variables below are not part of the informal specifica
-- tion given by SUN.
-- They are given a name, a type and a meaning that reflects a
-- part of the
-- system state, in order to be able to make a meaningful spe
-- cification. An
-- implementer of this class is naturally free to represent
-- the system state
-- with the help of other class and instance fields.
--
--   private byte[] valueExponent;
--   private byte[] valueModulus;
--   private boolean isInitExponent;
--   private boolean isInitModulus;


context RSAPrivateKey def:
   let valueExponent: Sequence(JByte)
   let valueModulus: Sequence(JByte)
   let isInitExponent: Boolean
   let isInitModulus: Boolean


context RSAPrivateKey inv:
      self.isInitExponent
      and
      self.isInitModulus
   implies
      self.isInitialized()


context RSAPrivateKey::setModulus(buffer: Sequence(JByte),
                                  offset: JShort,
                                  length: JShort)
  pre : buffer <> null
        and
        offset.asInt() >= 0
        and
        length.asInt() >= 0
        and
        offset.asInt()+length.asInt() <= buffer->size()
  post: (
           not excThrown(java::lang::Exception)
           and
```

```
           self.isInitModulus
           and
           (
                 not self.oclIsKindOf(javacardx::crypto::Key
Encryption)
                 or
                 self.getKeyCipher() = null
              implies
                 Util.arrayCompare(self.valueModulus, 0, buf
fer,
                                                      offset,
length) = 0
           )
        )
        or
        (
           excThrown(CryptoException)
           and
           CryptoException.systemInstance.getReason().asInt()
              = CryptoException.ILLEGAL_VALUE
        )


context RSAPrivateKey::setExponent(buffer: Sequence(JByte),
                                   offset: JShort,
                                   length: JShort)
  pre : buffer <> null
        and
        offset.asInt() >= 0
        and
        length.asInt() >= 0
        and
        offset.asInt()+length.asInt() <= buffer->size()
  post: (
           not excThrown(java::lang::Exception)
           and
           self.isInitExponent
           and
           (
                 not self.oclIsKindOf(javacardx::crypto::Key
Encryption)
                 or
                 self.getKeyCipher() = null
              implies
                 Util.arrayCompare(self.valueExponent, 0, buf
fer,
                                             offset, length) = 0
           )
        )
        or
        (
```

```
           excThrown(CryptoException)
           and
           CryptoException.systemInstance.getReason().asInt()
              = CryptoException.ILLEGAL_VALUE
        )


context RSAPrivateKey::getModulus(buffer: Sequence(JByte),
                                  offset: JShort): JShort
  pre : buffer <> null
        and
        offset.asInt() >= 0
        and
        offset.asInt() < buffer->size()
        and
        self.isInitialized()
  post: result.asInt() = self.valueModulus->size()
        and
        Util.arrayCompare(self.valueModulus, 0, buffer, off
set,
                                        self.valueModulus->si
ze()) = 0


context RSAPrivateKey::getExponent(buffer: Sequence(JByte),
                                   offset: JShort): JShort
  pre : buffer <> null
        and
        offset.asInt() >= 0
        and
        offset.asInt() < buffer->size()
        and
        self.isInitialized()
  post: result.asInt() = self.valueExponent->size()
        and
        Util.arrayCompare(self.valueExponent, 0, buffer, off
set,
                                        self.valueExponent-
>size()) = 0


endpackage
```

javacard.security.Signature

```
package javacard::security

...

-- abstract
context Signature::update(inBuff: Sequence(JByte),
                          inOffset: JShort,
                          inLength: JShort)
  pre : inBuff <> null
        and
        inOffset.asInt() >= 0
        and
        inLenght.asInt() >= 0
        and
        inOffset.asInt()+inLength.asInt() <= inBuff->size()
  post: not excThrown(java::lang::Exception)
        or
        (
           excThrown(CryptoException)
           and
          CryptoException.systemInstance.getReason().asInt()
              = CryptoException.UNINITIALIZED_KEY
        )


-- abstract
context Signature::sign(inBuff: Sequence(JByte),
                        inOffset: JShort,
                        inLength: JShort,
                        sigBuff: Sequence(JByte),
                        sigOffset: JShort): JShort
  pre : inBuff <> null
        and
        sigBuff <> null
        and
        inOffset.asInt() >= 0
        and
        inLength.asInt() >= 0
        and
        sigOffset.asInt() >= 0
        and
        inOffset.asInt()+inLength.asInt() <= inBuff->size()
  post: not excThrown(java::lang::Exception)
        or
        (
           excThrown(CryptoException)
           and
           (
```

```
               CryptoException.systemInstance.getRea
son().asInt()
                  = CryptoException.UNINITIALIZED_KEY
               or
               CryptoException.systemInstance.getRea
son().asInt()
                  = CryptoException.INVALID_INIT
               or
               CryptoException.systemInstance.getRea
son().asInt()
                  = CryptoException.ILLEGAL_USE
            )
         )

...

endpackage
```

javacardx.crypto.Cipher

```
package javacardx::crypto


context Cipher def:
   let key: Key
   let mode: JByte
   let algorithm: JByte
   let initialized: Boolean = false


context Cipher::getInstance(algorithm: JByte,
                            externalAccess: Boolean): Cipher
  pre : true
  post: (
           not excThrown(java::lang::Exception)
           and
           self.algorithm.asInt() = algorithm.asInt()
         )
         or
         (
           excThrown(javacard::security::CryptoException)
           and
           javacard::security::CryptoException.systemInstan
ce.getReason()
              = javacard::security::CryptoExcep
tion.NO_SUCH_ALGORITHM
         )


context Cipher::init(theKey: Key, theMode: JByte)
  pre : true
  post: (
           not excThrown(java::lang::Exception)
           and
           self.key = theKey
           and
           self.mode.asInt() = theMode.asInt()
           and
           self.initialized = true
         )
         or
         (
           excThrown(javacard::security::CryptoException)
           and
           javacard::security::CryptoException.systemInstan
ce.getReason()
              = javacard::security::CryptoExcep
tion.ILLEGAL_VALUE
```

```
        )


context Cipher::init(theKey: Key,
                     theMode: JByte,
                     bArray: Sequence(JByte),
                     bOff: JShort,
                     bLen: JShort)
  pre : true
  post: (
           not excThrown(java::lang::Exception)
           and
           self.key = theKey
           and
           self.mode.asInt() = theMode.asInt()
           and
           self.initialized = true
        )
        or
        (
           excThrown(javacard::security::CryptoException)
           and
           javacard::security::CryptoException.systemInstan
ce.getReason()
              = javacard::security::CryptoExcep
tion.ILLEGAL_VALUE
        )


context Cipher::getAlgorithm(): JByte
  pre : true
  post: result = self.algorithm


context Cipher::update(inBuff: Sequence(JByte),
                       inOffset: JShort,
                       inLength: JShort,
                       outBuff: Sequence(JByte),
                       outOffset: JShort): JShort
  pre : true
  post: (
           not excThrown(java::lang::Exception)
        )
        or
        (
           excThrown(javacard::security::CryptoException)
           and
           (
              (
                 javacard::security::CryptoException.systemIn
stance.getReason()
```

```
                    = javacard::security::CryptoExcep
tion.UNINITIALIZED_KEY
                 and
                 not self.key.isInitialized()
              )
              or
              (
                 javacard::security::CryptoException.systemIn
stance.getReason()
                    = javacard::security::CryptoExcep
tion.INVALID_INIT
                 and
                 not self.initialized
              )
              or
              javacard::security::CryptoException.systemIn
stance.getReason()
                 = javacard::security::CryptoExcep
tion.ILLEGAL_USE
           )
        )


context Cipher::doFinal(inBuff: Sequence(JByte),
                        inOffset: JShort,
                        inLength: JShort,
                        outBuff: Sequence(JByte),
                        outOffset: JShort): JShort
  pre : true
  post: (
           not excThrown(java::lang::Exception)
        )
        or
        (
           excThrown(javacard::security::CryptoException)
           and
           (
              (
                 javacard::security::CryptoException.systemIn
stance.getReason()
                    = javacard::security::CryptoExcep
tion.UNINITIALIZED_KEY
                 and
                 not self.key.isInitialized()
              )
              or
              (
                 javacard::security::CryptoException.systemIn
stance.getReason()
                    = javacard::security::CryptoExcep
tion.INVALID_INIT
```

```
                and
                not self.initialized
            )
            or
            javacard::security::CryptoException.systemIn
stance.getReason()
                = javacard::security::CryptoExcep
tion.ILLEGAL_USE
        )
    )


endpackage
```