Formal Specification and Verification of JavaCard's Application Identifier Class*

Joachim van den Berg, Bart Jacobs, Erik Poll

Dept. Computer Science, Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
{joachim, bart, erikpoll}@cs.kun.nl
http://www.cs.kun.nl/~{joachim, bart, erikpoll}

Abstract This paper discusses a verification in PVS of the `AID` (Application Identifier) class from the JavaCard API. The properties that are verified are formulated in the interface specification language JML. This language is also used to express the properties that are assumed about the native methods from the `Util` class that are used in the `AID` class. These properties include invariants for classes and behaviour specifications for methods; the latter give pre- and post-conditions describing the functional behaviour, and also specify when exceptions may be thrown.

1 Introduction

This paper describes the obvious next step after the formal specification of the JavaCard API [16,17] in the specification language JML [13,12], namely actual verification that the current reference implementation (version 2.1.1, [1]) satisfies these specifications. This verification is done with the proof tool PVS [14], using the translation of Java and JML to PVS, as incorporated in the LOOP tool [11,3]. This verification forms a test, both for the JML specifications, and for the Java implementations (and of course also for the LOOP tool). Here we concentrate on a small part of the API, involving essentially only the classes `Util` and `AID`. In an earlier version of this work only lightweight specifications were verified, but here more complete functional specifications will be studied. This is the first major case study in verification of JML specifications. (An earlier verification case study [8], on the `Vector` class from the Java API, did not use a formal semantics of JML.) The case study presented here heavily uses the tailor-made Hoare style logic developed especially for JML [10], in combination with dedicated proof strategies in PVS.

The JavaCard API specifications in JML make many implicit assumptions explicit, see [16,17], and provide very precise (and hopefully readable) documentation which should be useful for applet developers. But one can also make mistakes in writing specifications, so it is important to actually verify them, if possible. One option is to use so-called extended static checking, with ESC/Java [18],

* Extended version of [4]. To appear in: I. Attali and Th. Jensen, editors, *Proceedings of the Java Card 2000 Workshop* (Springer LNCS 2001)

which can automatically spot potential violations of specifications. This is certainly very effective and useful, but it can only deal with relatively “simple” properties—which do not include the properties with universal quantification used in this paper; and though ESC/Java can detect many errors it never guarantees the absence of any errors. We see our approach as complementary, providing additional certainty: our verifications are based on a mathematical semantics of Java and JML (expressed in the logic of PVS) that allows in principle arbitrarily complex assertions to be used in specifications. But proving such assertions is interactive, and far from automatic. The fact that both automatic checking using ESC/Java and interactive verification using PVS are possible for JML is a strong advantage of this specification language. Additionally, it is possible to do run-time checking of JML assertions [5].

This paper cannot give more than an impression of some of the issues in API specification and verification. It does include the full JML specifications of the `AID` class (that have been verified) in Section 3, and also the specifications of the two relevant (native) methods from the `Util` class (in Section 2) that are used in `AID`. Section 4 highlights some of the interesting issues that came up in the specification and verification of the `AID` class; in particular it discusses exceptions and invariants, and the important role they play in the verification. Finally, Section 5 draws some conclusions.

2 JavaCard's `Util` class: specification

The `AID` class makes use of two methods from the `Util` class, namely `arrayCopy` and `arrayCompare`. Both these methods are static, final and native: ‘static’ means that they can be invoked without a receiving object, ‘final’ that they cannot be overridden, and ‘native’ that no actual Java code is provided, but their implementation is given in some other (low-level) language.

The behaviour of these methods is described in informal comments in the `Util.java` file [1]. We quote these explanations:

`arrayCopy`: “Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array (non-atomically).”

`arrayCompare`: “Compares an array from the specified source array, beginning at the specified position, with the specified position of the destination array from left to right. Returns the ternary result of the comparison : less than (-1), equal(0) or greater than(1).”

More information is given in the informal (but very detailed) specifications included as javadoc comments in the code [1], especially about when exceptions may be expected. Such information is important for correct use of the JavaCard API in applets.

We shall use the formal specifications in JML given in Figures 1 and 2 (see also [16]), which closely follow these informal specifications. What we add are

```
/*@ normal_behavior
  @   requires: src != null &&  srcOff >= 0 &&
  @               srcOff+length <= src.length &&
  @             dest != null && destOff >= 0 &&
  @               destOff+length <= dest.length &&
  @             length >= 0;
  @ modifiable: \nothing;
  @    ensures: \result == -1 || \result == 0 || \result == 1;
  @    ensures: \result == 0
  @              <==>
  @             \forall (byte i) 0 <= i && i < length
  @                              ==> src[srcOff+i] == dest[destOff+i];
  @ also
  @ behavior
  @   requires: true;
  @ modifiable: \nothing;
  @    signals: (NullPointerException)
  @                src == null || dest == null;
  @    signals: (ArrayIndexOutOfBoundsException)
  @                (src != null &&
  @                  (0 > srcOff || srcOff + length > src.length))
  @                ||
  @                (dest != null &&
  @                  (0 > destOff || destOff + length > dest.length));
  @*/
public static final native byte arrayCompare(byte[] src,
                                             short srcOff,
                                             byte[] dest,
                                             short destOff,
                                             short length)
throws ArrayIndexOutOfBoundsException,
       NullPointerException;
```

Figure1. JML specification for `arrayCompare` from `Util`

```
/*@ behavior
  @   requires: src != null &&  srcOff >= 0 &&
  @               srcOff+length <= src.length &&
  @             dest != null && destOff >= 0 &&
  @               destOff+length <= dest.length &&
  @             length >= 0;
  @ modifiable: dest[destOff..destOff+length-1],
  @             TransactionException.systemInstance.reason;
  @    ensures: \forall (short i) 0 <= i && i < length
  @                  ==> dest[destOff+i] == \old(src[srcOff+i]);
  @    signals: (TransactionException e)
  @                 e.getReason() == TransactionException.BUFFER_FULL;
  @ also
  @ behavior
  @   requires: true;
  @ modifiable: dest[destOff..destOff+length-1],
  @             TransactionException.systemInstance.reason;
  @    signals: (TransactionException e)
  @                 e.getReason() == TransactionException.BUFFER_FULL;
  @    signals: (NullPointerException)
  @                 src == null || dest == null;
  @    signals: (ArrayIndexOutOfBoundsException)
  @                (src != null &&
  @                  (0 > srcOff || srcOff + length > src.length))
  @                ||
  @                (dest != null &&
  @                  (0 > destOff || destOff + length > dest.length));
  @*/
public static final native short arrayCopy(byte[] src,
                                           short srcOff,
                                           byte[] dest,
                                           short destOff,
                                           short length)
throws ArrayIndexOutOfBoundsException,
       NullPointerException,
       TransactionException;
```

Figure2. JML specification for `arrayCopy` from `Util`

the explicit distinction between normal and exceptional termination, plus the precise information about items that may be modified by the methods.

Instead of explaining JML in general, we explain only the meaning of these specifications. The `behavior` keyword indicates that if the precondition as given by the `requires` clause holds, then either the method terminates normally, and the "normal" postcondition after the `ensures` keyword holds, or the method terminates abruptly because of an exception of the type indicated after `signals`, and the ensuing "abrupt" postcondition holds. The keyword `normal_behavior`, instead of just `behavior`, indicates that the method must terminate normally if the precondition holds. The `behavior` specifications are translated into *partial* Hoare formulas, and the `normal_behavior` specifications into *total* Hoare formulas, in a special version of Hoare logic adapted to JML [10].

The `modifiable` clauses tell which items may be changed by the method, and thus also, implicitly, which are left unaltered. The method `arrayCompare` does not modify anything, and thus has no side-effect. The `arrayCopy` method can modify certain entries of its parameter array `dest`, namely those in the range `offset`, ..., `offset + length - 1`. The need for `TransactionException.-systemInstance.reason` is discussed in Section 4.2. The `TransactionException` may occur when an overflow arises in JavaCard's transaction buffer—which is used to enable rollback of operations in case of failure. The `ensures` clauses describe the postconditions, and thus the functional behaviour of these methods. Notice that we did not describe everything: it is not made explicit what it means when the result of `arrayCompare` is -1 or 1. This is possible, but cumbersome and these cases are probably not often needed. The various signals clauses of the form `signals: (E e) Q` express that when an exception `e` is thrown which is an instance of (exception) class `E`, then the postcondition `Q` must hold. These are useful for distinguishing the various possible causes for abrupt termination.

A subtle point is the use of `\old(...)` in the `ensures` clause of the `arrayCopy` method. This is needed to allow for aliasing, *i.e.* for `src == dest`. In this case `src` may be modified, and so we have to refer to the original entries of the `src` array to explain the copying. The informal javadoc specifications explicitly state that this is what happens if `src` and `dest` are aliases.

3 JavaCard's `AID` class: specification and verification

In the JavaCard platform each applet instance and package is uniquely identified and selected by a so-called application identifier (AID)—and not, as usual in Java, by a string possibly in combination with a domain name, see [6, §§3.8]. AIDs are used in loading and linking. As prescribed in the ISO 7816 standard, each AID consists of an array of bytes, ranging in length from 5 to 16. The first five bytes form what is called the resource identifier (RID), which is assigned by ISO to a company. The possible remainder (between 0 and 11 bytes) forms the proprietary identifier extension (PIX), which is under the control of individual companies.

Before giving the specification of the `AID` class together with the current implementation[1] we describe the essentials of the `AID` class. Its field, constructor and five methods, without their implementations, look as follows.

```
public final class AID {
  byte[] theAID;
  public AID(byte[] bArray, short offset, byte length);
  public byte getBytes(byte[] dest, short offset);
  public boolean equals(Object anObject);
  public boolean equals(byte[] bArray, short offset, byte length);
  public boolean partialEquals(byte[] bArray, short offset, byte length);
  public boolean RIDEquals (AID otherAID);
}
```

Our JML specification of this class adds a class invariant, a history constraint and pre-/post-conditions for its constructor and methods.

Unlike the pre-/post-conditions, the invariant and history constraint are not explicitly stated in the informal documentation, but especially the invariant is crucially needed to prove that the implementation meets the specification. Since the array `theAID` of an AID consists of a 5-byte RID possibly together with a PIX of up-to 11 bytes, our invariant is

```
/*@
  @ invariant: theAID != null &&
  @            5 <= theAID.length && theAID.length <= 16;
  @*/
```

The proof obligation is to show that this property holds after normal termination of the constructor, and also that it holds after termination (both normal and abrupt) of each method, assuming it holds in the state in which the method is invoked[2].

Another class assertion is a so-called constraint:

```
/*@
  @ constraint: theAID == \old(theAID);
  @*/
```

It says that the `theAID` field never changes. Again, this property should be established for all methods. Class invariants and constraints are included in the semantics of (pre- and post-conditions of) methods specifications.

The `AID` class with JML specifications looks as follows.

```
public final class AID {
/*@
```

[1] The implementation has revision number 1.19, written by Ravi, under Sun Microsystems copyright (1999)

[2] What is surprising is that the `theAID` field is not declared as `private`. As it stands, it can be modified from the outside (but only within its package), making it vulnerable to a breach of the invariant. We consider the omission of the `private` access modifier a bug.

```
  @  invariant: theAID != null &&
  @             5 <= theAID.length && theAID.length <= 16;
  @
  @ constraint: theAID == \old(theAID);
  @*/

 byte[] theAID;

/*@ behavior
  @   requires: 5 <= length && length <= 16 &&
  @             bArray != null &&
  @             0 <= offset && offset + length <= bArray.length;
  @ modifiable: theAID, theAID[*],
  @             TransactionException.systemInstance.reason;
  @    ensures: length == theAID.length &&
  @             (\forall (short index) 0 <= index && index < length ==>
  @                    theAID [index] == bArray [offset + index]);
  @    signals: (TransactionException e)
  @                 e.getReason() == TransactionException.BUFFER_FULL;
  @*/
public AID( byte[] bArray, short offset, byte length )
    throws SystemException {
  if (length < 5 || length > 16)
    SystemException.throwIt(SystemException.ILLEGAL_VALUE);
  theAID = new byte[length];
  Util.arrayCopy( bArray, offset, theAID, (short)0, length );
}

/*@ behavior
  @   requires: dest != null && dest != theAID &&
  @             0 <= offset &&
  @             offset + theAID.length <= dest.length;
  @ modifiable: dest[offset..offset+theAID.length-1],
  @             TransactionException.systemInstance.reason;
  @    ensures: (\forall (short index)
  @                  0 <= index && index < theAID.length ==>
  @                  dest [offset + index] == theAID [index]) &&
  @             \result == theAID.length;
  @    signals: (TransactionException e)
  @                  e.getReason() == TransactionException.BUFFER_FULL;
  @ also
  @ behavior
  @   requires: true;
  @ modifiable: dest[offset..offset+theAID.length-1],
  @             TransactionException.systemInstance.reason;
  @    signals: (TransactionException e)
  @                 e.getReason() == TransactionException.BUFFER_FULL;
  @    signals: (NullPointerException)
  @                 dest == null;
  @    signals: (ArrayIndexOutOfBoundsException)
```

```
  @                 dest != null &&
  @                 (0 > offset || offset + theAID.length > dest.length);
  @*/
public byte getBytes (byte[] dest, short offset) {
  Util.arrayCopy( theAID, (short)0, dest, offset, (short)theAID.length );
  return (byte) theAID.length;
}

/*@ normal_behavior
  @   requires: (anObject instanceof AID) ==> \invariant_for((AID)anObject);
  @ modifiable: \nothing;
  @    ensures: \result ==
  @             (anObject instanceof AID &&
  @             ((AID)anObject).theAID.length == theAID.length &&
  @             (\forall (short index) 0 <= index && index < theAID.length
  @                ==> theAID [index] == ((AID)anObject).theAID [index]));
  @*/
public boolean equals( Object anObject ) {
  if ( !(anObject instanceof AID) ||
        ((AID)anObject).theAID.length != theAID.length) return false;
  return (Util.arrayCompare(((AID)anObject).theAID, (short)0, theAID,
                            (short)0, (short)theAID.length) == 0);
}

/*@ normal_behavior
  @   requires: bArray == null || (0 <= offset &&
  @                                length >= 0 &&
  @                                offset + length <= bArray.length
  @              /// the extra condition below is needed to prove that
  @              /// the (apparently unnecessary) assignment on testByte
  @              /// never fails.
  @              /// The 3 conjuncts above are not strong enough.
  @                                && offset + length >= 1);
  @ modifiable: \nothing;
  @    ensures: \result ==
  @             (bArray != null &&
  @              length == theAID.length &&
  @              (\forall (short index) 0 <= index && index < length ==>
  @                  theAID [index] == bArray [offset + index]));
  @ also
  @ behavior
  @   requires: bArray != null;
  @ modifiable: \nothing;
  @    signals: (ArrayIndexOutOfBoundsException)
  @                offset + length < 1 ||
  @                0 > offset          ||
  @                length < 0          ||
  @                offset + length > bArray.length;
  @*/
public boolean equals( byte[] bArray, short offset, byte length ) {
```

```
  if (bArray == null) return false;
  // verify the array index
  byte testByte = bArray[(short)(offset + length - (short)1)];
  return ((length == theAID.length) &&
     (Util.arrayCompare(bArray, offset, theAID, (short)0, length) == 0));
}

/*@ normal_behavior
  @   requires: bArray == null || (0 <= offset &&
  @                               length >= 0 &&
  @                               offset + length <= bArray.length);
  @ modifiable: \nothing;
  @    ensures: \result ==
  @             (bArray != null &&
  @              length <= theAID.length &&
  @              (\forall (short index) 0 <= index && index < length ==>
  @                 theAID [index] == bArray [offset + index]));
  @ also
  @ behavior
  @   requires: bArray != null && length <= theAID.length;
  @ modifiable: \nothing;
  @    signals: (ArrayIndexOutOfBoundsException)
  @                0 > offset ||
  @                length < 0 ||
  @                offset + length > bArray.length;
  @*/
public boolean partialEquals( byte[] bArray, short offset, byte length ) {
  if ( bArray == null || length > theAID.length ) return false;
  return (Util.arrayCompare(bArray, offset,
                            theAID, (short)0, length) == 0);
}

/*@ normal_behavior
  @   requires: true; /// implicitly: \invariant_for(otherAID)
  @    ensures: \result ==
  @             (otherAID != null &&
  @             (\forall (short index) 0 <= index && index < 5 ==>
  @                    theAID [index] == otherAID.theAID [index]));
  @*/
public boolean RIDEquals ( AID otherAID ) {
  if ( otherAID == null ) return false;
  if ( Util.arrayCompare( theAID, (short)0, otherAID.theAID,
                          (short)0, (short)5 ) == 0 ) return true;
  return false;
}
}
```

The LOOP tool translates the Java implementations to certain (complicated) structures describing their semantics in PVS. The JML specifications become predicates on such structures. The actual verification task then consists of showing that the structures arising from the given Java implementation satisfy these predicates. For the methods that are called in these implementations (like `arrayCompare` or `getReason`) their JML specifications are used, and not their Java implementations (if any). In this way, all of the above specifications have been proven in PVS, for the given implementations. The proofs are done interactively, by using the JML proof rules from [10]—following the structure of the (translated) Java code—together with appropriate powerful proof strategies.

On average, constructing such a proof takes about one hour—assuming the specification is correct. But during this case study we frequently had to go back and adjust a minor detail in the specifications. It is surprisingly hard to get specifications precisely right. This fact makes actually doing the verifications so important.

4 Some special issues

Due to space restrictions we cannot discuss all aspects of the `AID` specification and verification, so we concentrate on several special points.

4.1 `behavior` and not `exceptional_behavior`

Our `Util` and `AID` specifications contain `normal_behavior` (for total correctness) and `behavior` (for partial correctness, when exceptions may be thrown). When specifying under which conditions methods throw certain exceptions as we have done using `behavior` specifications, one has to be very careful with the subtle difference between specifying when a method *may* throw an exception and specifying when a method *must* throw an exception, especially when there is the possibility of throwing more than one exception. The `signals`-clauses in the `behavior` specifications state that an exception *may* be thrown (and that something special condition is then the case). In the JavaCard API documentation it is explicitly stated that when different kinds of exceptions may be thrown, then there are no guarantees about which one will actually arise if conditions overlap. This excludes `exceptional_behavior`, because it is used for specifying that an exception *must* be thrown if a given precondition is met. In our experience it is extremely tricky to specify when a certain exception *must* be thrown, and, moreover, typically one is not so interested in such properties.

4.2 No Exception object creation

Because of the limited resources available on smart cards, exception objects are not created during run-time [6, §§6.2]. Instead, an already existing static exception object, typically called `systemInstance`, is thrown via the `throwIt` method, see the `AID` constructor. The different exceptions are distinguished via a `reason` field. That is why there are so many (qualified) `reason`'s in the `modifiable` clauses: any method that can throw a `TransactionException` can modify the `reason` field of the static exception object `TransactionException.-systemInstance`.

(Since static initialization does not take place during the execution of JavaCard programs, we have a semantics which is different from Java programs. In [2] it is described how the underlying memory model in PVS involves for each class with static initialization a special Boolean telling whether static initialization for this class has already taken place. Whenever a member from this class is used, the value of this Boolean is checked. We have disabled such a check for JavaCard programs, and thereby disabled static initialization.)

4.3 Class invariants

The preservation of invariants in an object oriented setting is a notoriously complicated issue, see for instance [7,9,19,15]. The hardest problem is caused by aliasing: an object can hold references to other objects and its invariant can depend on the states of these other objects, and vice versa. Another problem is that of callbacks, also known as re-entrance.

Ideally, the way to deal with invariants is to assume that *all* objects—not just `this`—satisfy their invariants at beginning of a method body, and to prove that *all* objects satisfy their invariants at the end of the method body[3]. Invariants can then be temporarily broken during the method invocation. However, note that if our method body contains a method invocation then all invariants should be re-established at this point, because this is required by the precondition of this method that is invoked.

However, a fundamental problem with this idealized approach above is how to prove that a method of an object does not break the invariants of some other objects. For example, the method `getBytes` changes the contents of the byte array `dest` it gets as an argument; how can we possibly prove that this does not break the invariant of some other object that also holds a reference to this array? To do this we would have to know all other objects and class invariants that exist. It should be clear that this approach is not really feasible.

In light of this problem, what we currently do when verifying methods in PVS is that we assume at the beginning of the method body

- that the precondition holds,
- that `this` satisfies its invariant, and
- that any objects the method receives as parameters (if any) satisfy their invariants;

then we prove

- that the postcondition holds,
- that the invariant of `this` hold at the end of the method body, and

[3] For private methods we can drop the requirement that invariants are maintained, but, since all methods of `AID` are public, we ignore this here.

– that the appropriate invariant for a result or exception object, if any, holds[4].

What actually happens is that the LOOP tool strengthens all pre-conditions with the assertion that `this` satisfies its invariant, and with assertions that the parameter (if any) satisfy their invariants. So, for example, the pre-condition of `RIDEquals` is strengthened to include an assertion that its parameter `otherAID` of class `AID` satisfies the `AID`-invariant. This can then be used during verification of the method. (And in fact this is needed in the verification, namely to establish that `otherAID.theAID` is not null and has a length of at least 5 in the invocation of `arrayCompare` in `RIDEquals`.)

In a dual fashion, when a method specification is *used*, to prove a property of some invocation of the method, then this strengthened pre-condition has to be established; this includes proving that all parameters meet their invariants.

However, this implicit strengthening of preconditions does not suffice for all cases, and sometimes preconditions have to be strengthened further, explicitly. For example, if we have an object `obj` of class `Object`, and (down)cast it to an arbitrary class `A`, then we would like to use that `(A)obj` satisfies the `A`-invariant, and not just the `Object`-invariant. This actually comes up during verification of the `equals` method from `AID` with parameter `Object anObject`. After a check that the parameter belongs to class `AID`, it is used as an `AID` object, and the `AID`-invariant must be assumed for it. At the moment this problem is dealt with manually by adding an extra clause in the `requires` clause, namely

```
(anObject instanceof AID) ==> \invariant_for((AID)anObject)
```

which states that if `anObject` is an instance of class `AID`, then is satisfies the `AID`-invariant. This can then be used to prove the method correct, but of course has to be established for any invocation of `equals`. How to handle this in general still has to be elaborated.

4.4 Class invariants and exceptions

Our semantics of JML requires that invariants are established by every normally terminating constructor and maintained by every method, regardless of whether the method terminates normally or terminates abnormally by throwing an exception. Invariants (for `this`) are implicitly included in all postconditions, both in the `ensures` and in the `signals` clauses, of all methods. Requiring that invariants are maintained even if an exception is thrown may seem an overly strong requirement. However, it is needed, because if the exception is caught later and normal execution of the program resumes, then the invariants should still hold. Inside method bodies the invariants may be temporarily broken; care must be taken that no exception can be thrown while this is the case.

For constructors the situation is more subtle, because the invariant typically does not hold until right at the end of the constructor body. But if an exception

[4] In the end, the invariants of the parameters should still hold, if all methods applied to them preserve invariants—as they should.

is thrown, one cannot retrieve a partially constructed object via catching. So there is no need to require the invariant for abrupt termination. Therefore, only the `ensures` clauses are interesting for constructor specifications. For the `AID` constructor we do not write a `normal_behavior` instead of a `behavior` specification because the `Util.arrayCopy` method may still throw a `Transaction` exception (when the transaction buffer becomes full). Trying to exclude this in the precondition is not so easy.

5 Conclusions

We have sketched the verification of the formal JML specification for the `AID` class in the JavaCard API using the theorem prover PVS, based on the semantics of Java and JML provided by the LOOP tool. In the end, what we have obtained is a fully verified formal specification of a small part of the JavaCard API. This is of course not a major example, but it does show the possibility of such validation efforts for class libraries.

One interesting—and sometimes frustrating—aspect of doing this kind of formal verification is the continuing discovery of ever more subtle mistakes in the formal specification. The verification of an individual method typically involves dealing explicitly with many possible cases, some of which are easily overlooked. Indeed, everything in Java can throw an exception, which creates many possibilities to be explored. But handling all of them is important, because unexpected exceptions form a notorious cause of (run-time) errors in programs.

Work on formally specifying and verifying more classes of the JavaCard API is continuing[5]. Our aim is to obtain a fully verified formal specification of the JavaCard API that can serve as the basis for actual applet verification. Initial goals for these applet verifications will be proving the absence of runtime exceptions and non-termination, and proving the absence of unwanted side effects (notably side effects on other applets) expressed by `modifiable` clauses. Achieving all this will still require the development of more proof rules and PVS tactics to further reduce the effort required for verifications.

References

1. JavaCard API 2.1. `http://java.sun.com/products/javacard/htmldoc/`.
2. J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert, C. Choppy, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques*, number 1827 in Lect. Notes Comp. Sci., pages 1–21. Springer, Berlin, 2000.
3. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. Techn. Rep. CSI-R0019, Comput. Sci. Inst., Univ. of Nijmegen. To appear at TACAS'01., 2000.

[5] The work on the JavaCard API is part of European IST-project "VerifiCard", see `www.verificard.org`.

4. J. van den Berg, B. Jacobs, and E. Poll. Formal specification and verification of JavaCard's Application Identifier Class. Techn. Rep. CSI-R0014, Comput. Sci. Inst., Univ. of Nijmegen. Appeared in: Proceedings of the JavaCard Workshop, Cannes. INRIA Techn. Rep. Sept. 1999.
5. A. Bhorkar. A run-time assertion checker for Java using JML. Techn. Rep. 00-08, Dep. of Comp. Science, Iowa State Univ. (`http://www.cs.iastate.edu/~leavens/JML.html`), 2000.
6. Z. Chen. *Java Card Technology for Smart Cards*. The Java Series. Addison-Wesley, 2000.
7. D. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Technical report, Compaq Systems Research Center, Palo Alto, 1998. Research Report 156.
8. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. Techn. Rep. CSI-R0007, Comput. Sci. Inst., Univ. of Nijmegen. To appear in *Software Tools for Technology Transfer*, 2001.
9. K. Huizing, R. Kuiper, and SOOP. Verification of object oriented programs using class invariants. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in Lect. Notes Comp. Sci., pages 208–221. Springer, Berlin, 2000.
10. B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. Techn. Rep. CSI-R0018, Comput. Sci. Inst., Univ. of Nijmegen. To appear at FASE'01., 2000.
11. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 329–340. ACM Press, 1998.
12. G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov and B. Rumpe, editors, *Behavioral Specifications of Business and Systems*, pages 175–188. Kluwer, 1999.
13. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Techn. Rep. 98-06, Dep. of Comp. Sci., Iowa State Univ. (`http://www.cs.iastate.edu/~leavens/JML.html`), 1999.
14. S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.
15. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S.D. Swierstra, editor, *Programming Languages and Systems*, number 1576 in Lect. Notes Comp. Sci., pages 162–176. Springer, Berlin, 1999.
16. E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Smart Card Research and Advanced Application*, pages 135–154. Kluwer Acad. Publ., 2000.
17. E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the JavaCard API in JML: the APDU class. *Comp. Networks Mag.*, 2001. To appear.
18. Extended static checker ESC/Java. Compaq System Research Center. `http://www.research.digital.com/SRC/esc/Esc.html`.
19. C. Szyperski. *Component Software*. Addison-Wesley, 1998.