# Specification and Verification of the JavaCard API


Bart Jacobs

Dep. Comp. Sci, Univ. Nijmegen

bart@cs.kun.nl

http://www.cs.kun.nl/~bart

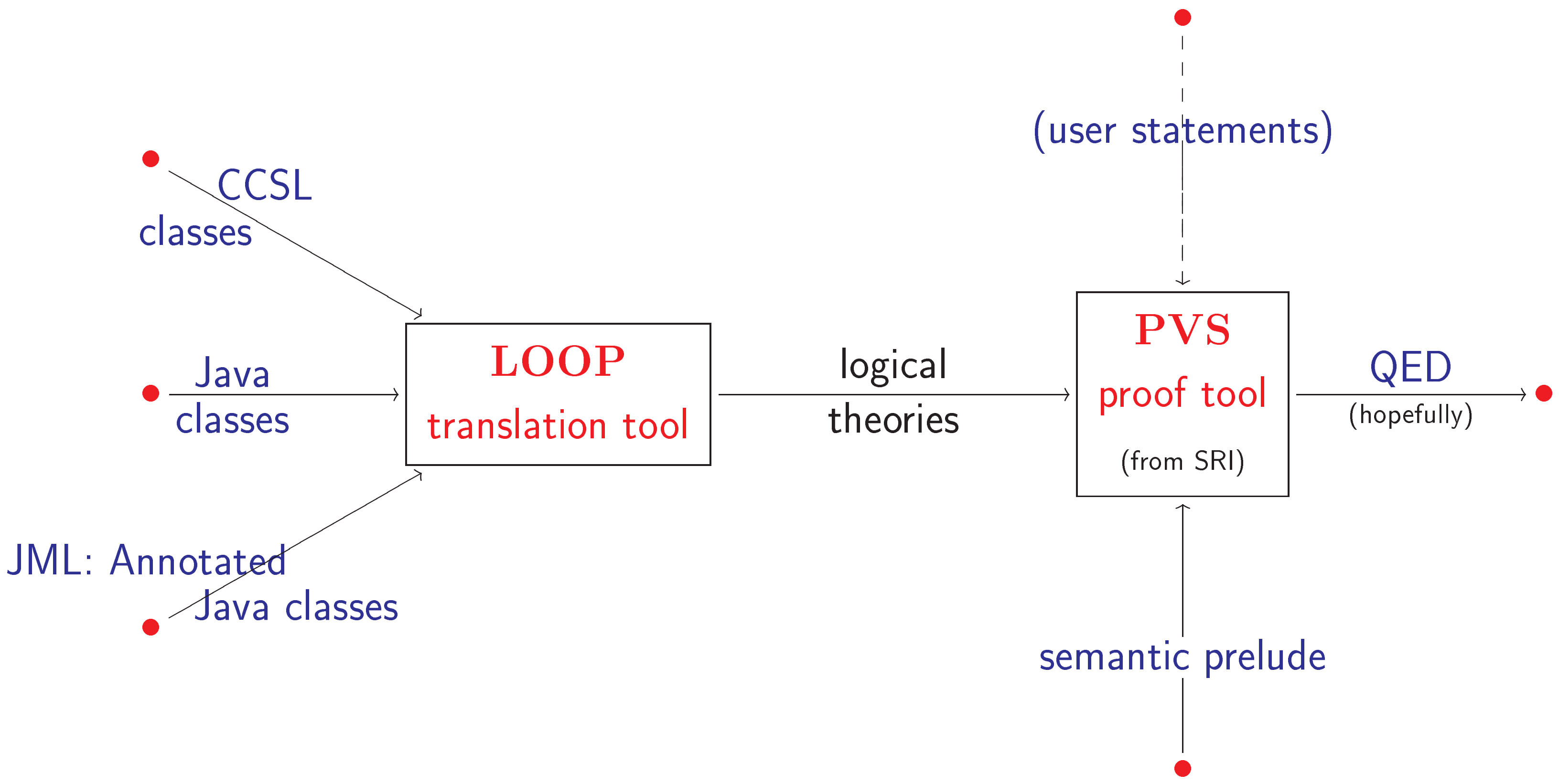

Joint work with:


Joachim van den Berg

Erik Poll

## Contents

- The LOOP project: Logic of Object-Oriented Programming
- JavaCard: simplified Java for smart cards
- JML: a behavioural interface specification language for Java
- Basics of JML semantics
- JavaCard API specification in JML
- Verification of the current (reference) implementation
- Related work & Conclusions




The LOOP tool is implemented in Ocaml and has an Emacs interface.

It is possible to use also Isabelle as a back-end theorem prover.

## LOOP essentials

A fresh attack on the old goal of program verification via a combination of:

- modern semantical theories, using coalgebras to model state-based (object-oriented) systems;
- powerful tools for translating (LOOP) and reasoning (PVS/Isabelle);
- a formal specification language JML that grew out of experience with Larch/C++ and builds on the design-by-contract approach from Eiffel.
- application to real (non-idealised) programming language (namely Java and JavaCard), with all their messy details.

### Note

This is not meta-level reasoning about Java as a language (investigating e.g. typesafety, or bytecode verification), but about concrete Java programs.

## Smart card:

Credit card-sized plastic card with embedded chip, capable of storing & processing (notably encrypting) data. Used as:

- bank cards
- phone cards
- SIM's in mobile phones or other handheld devices
- access or authentication cards, e.g. for electronic commerce
- ...

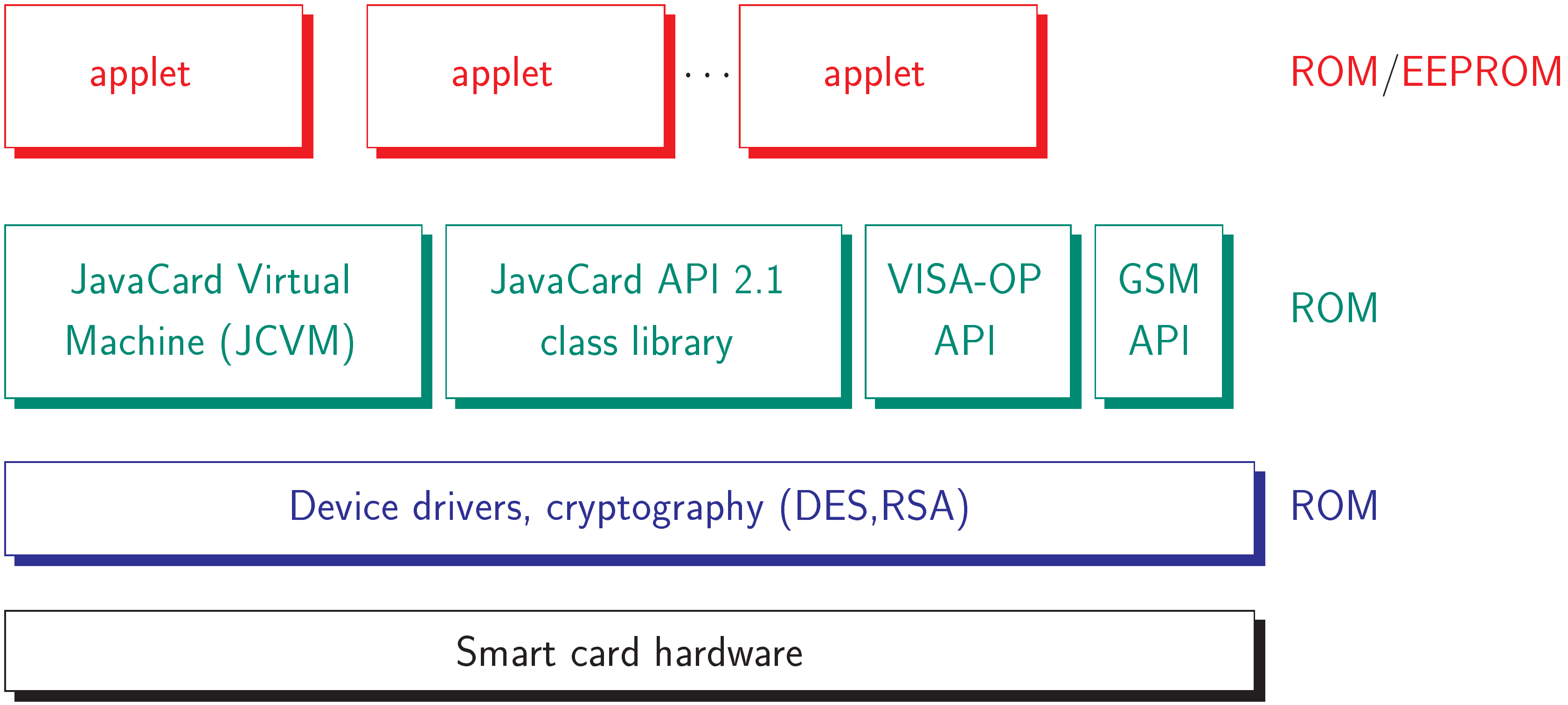
## JavaCard:

Simplified version of Java for programming the new generation of smart cards. It has no floats, no threads, ... , because of hardware constraints:

- RAM: 256 Bytes - 4 KBytes
- ROM: 16 - 64 KBytes
- EEPROM: 4 - 64 KBytes

No byte-code verification of application programs (applets), but code-signing after off-card verification.

# JavaCard smart card architecture



## Advantages of "open" smart cards, like with JavaCard

- Independence of underlying hardware
- High-level language for applets: everyone can write his/her own applets
- Post-issuance: downloading new applets after card has been issued (mobile code)
- Multi-application: multiple applets on the same smart card.

## Disadvantages

- Security, both of the platform and of individual applets.

  Imagine a virus on your smart card: a malicious applet exploiting some weakness of the platform, of the off-card byte code verification, or of other applets.

## Applet Verification

- is important, given the sensitivity of the applications
- seems feasible, given the limited size & complexity of the language and the platform
- much wanted by the card issuers, in order to protect their products, services and their image.

Our starting point: formal specifications for the JavaCard API, based on existing informal specifications.

Currently, the API is bout 45 classes, of which 20 trivial.

We use JML as formal specification language.

(There is also platform verification dealing for example with the JavaCard virtual machine and run-time environment; it is a different topic.)

# JML: Java Modeling Language

JML is a behavioural interface specification language, developed by Gary Leavens et al at Iowa State Univ, in collaboration with the ESC/Java group at Compaq SRC, and the LOOP group in Nijmegen.

In JML, Java programs can be annotated with assertions, that give

- pre- and postconditions (like in Eiffel)
- invariants and constraints
- specification-only variables
- frame conditions (modifiability constraints)
- ...

These annotations are written as special comments: `/*@ ... @*/`

Pre-/post-conditions, invariants and constraints in JML are Java boolean expressions, extended with special operators like `\forall`, `\exists`, `\old( )`, ...

Goal: JML should be usable by Java programmers, without assuming mathematical training.

# JML Example

```
public int sqrt (int n)
   /*@ normal_behavior
     @   requires: n >= 0;
     @   ensures: \result * \result <= n &&
     @              n < (\result+1) * (\result+1);
     @*/
```

Pre- and postconditions form a contract between an object and its clients:

- client must ensure precondition, and may assume postcondition
- object may assume precondition, and must ensure postcondition

as in Eiffel and Design by Contract, and UML (OCL).

## Weaker specs for `sqrt`:

```
public int sqrt (int n)
  /*@ normal_behavior
    @   requires: n >= 0;
    @   ensures: \result >= 0;
    @*/
```

or

```
public int sqrt (int n)
  /*@ normal_behavior
    @   requires: n >= 0;
    @   ensures: true;
    @*/
```

At this stage, our JML specifications for the JavaCard API are such lightweight (incomplete, weak) specs.

These specs only describe

- preconditions that rule out "bad" behaviour ("bad" termination, i.e. unwanted exceptions)
- the weakest postconditions we can get away with (nearly always `"true"`): no functionality

Such weak specifications are useful because they make many implicit assumptions explicit.

# Logic versus Programming

## Spectrum:

| Logic | | Programming |
|---|---|---|
| Denotational | | Operational |

Most specification languages lean towards the left.

But JML tries to be in the world of programmers, and is situated towards the right. This creates certain tensions in its semantics.

## JML is a superset of Java

Within the LOOP project a (coalgebraic) semantics for Java has already been developed, and used in many case study verifications. This same semantics will be used for JML.

## Semantics of Java expressions

Recall that Java (and hence JML) expressions can hang, terminate normally, or terminate abruptly (because of an exception). We describe them as coalgebras acting on a state space Self:

$$\mathsf{Self} \longrightarrow 1 + (\mathsf{Self} \times T) + (\mathsf{Self} \times \mathsf{Excp})$$

where $T$ is the type of outputs, and Excp is the type of exceptions. We use the following constructors for the three output options, in the interpretation of a method $m$:

$$[\![ m ]\!] \cdot x = \begin{cases} \mathsf{hang}() \\ \mathsf{norm}(x', t) \\ \mathsf{abnorm}(x', e) \end{cases}$$

## First approximation: ordinary predicates

For a method $m$ of result type $T$,
a precondition $P\colon$ Self $\rightarrow$ bool, and
a postcondition $Q\colon T \rightarrow$ Self $\rightarrow$ bool,
a specification for $m$ of the form:

```
normal_behavior
    requires:  P;
    ensures:   Q;
```

is interpreted as

$$\forall x\colon \mathsf{Self}.\ P \cdot x \implies \begin{array}{l} \mathsf{CASES}\ [\![\, m \,]\!] \cdot x\ \mathsf{OF}\ \{ \\ \quad \mid \mathsf{hang}() \mapsto \mathsf{false} \\ \quad \mid \mathsf{norm}\,(x', t) \mapsto Q \cdot t \cdot x' \\ \quad \mid \mathsf{abnorm}\,(x', e) \mapsto \mathsf{false}\, \} \end{array}$$

or as a total Hoare triple

$$[\, P \,]\, m\, [\, Q \,]$$

# Exceptions in JML predicates 1

## Example

Consider the following fragment:

```
public static int firstElement (int[] array)
   /*@ normal_behavior
     @   requires : array.length > 0;
     @    ensures : true;
     @*/
   { return array[0]; }
```

Note: there is a problem if the paremeter `array` is a null-reference: the pre-condition then throws a `NullPointerException`.

# Exceptions in JML predicates 2

## What should this mean?

Should such a pre-condition throwing an exception be interpreted as true or as false?

And what about exceptions in post-condtions? Or in invariants?

Of course, the specifier should have used as pre-condition:

```
...
requires: array != null && array.length > 0
...
```

But it is easy to make such mistakes.

Our JML semantics should be able to handle such predicates. A clear choice is needed.

# From JML- to ordinary predicates 1

There are various ways to translate

$$\boxed{\begin{array}{c}\text{JML-predicates:}\\ \mathsf{Self} \longrightarrow 1 + (\mathsf{Self} \times \mathsf{bool}) + (\mathsf{Self} \times \mathsf{Excp})\end{array}} \longmapsto \boxed{\begin{array}{c}\text{ordinary predicates:}\\ \mathsf{Self} \longrightarrow \mathsf{bool}\end{array}}$$

For example:

$$\left.\begin{array}{c}\text{hanging}\\ +\\ \text{abrupt termination}\end{array}\right\} \longmapsto \left\{\begin{array}{l}\text{false}\\ \quad\text{or}\\ \text{true}\\ \quad\text{or}\\ \text{arbitrary boolean}\end{array}\right.$$

Note that side-effects are ignored in the translation.
(But side-effects should be avoided in assertions anyway.)

## From JML- to ordinary predicates 2

We choose the first option, and formalise it via a function BE2B:

$$
\begin{array}{l}
p\colon \mathsf{Self} \longrightarrow 1 + (\mathsf{Self} \times \mathsf{bool}) + (\mathsf{Self} \times \mathsf{Excp}) \;\vdash \\
\qquad \mathsf{BE2B} \cdot p \;:\; \mathsf{Self} \longrightarrow \mathsf{bool} \;\stackrel{\mathsf{defn}}{=} \\
\qquad\qquad \lambda x\colon \mathsf{Self}.\;\; \mathsf{CASES}\; p \cdot x \;\mathsf{OF}\; \{ \\
\qquad\qquad\qquad\qquad\qquad \mid \mathsf{hang}() \mapsto \mathsf{false} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathsf{norm}(x', b) \mapsto b \\
\qquad\qquad\qquad\qquad\qquad \mid \mathsf{abnorm}(x', e) \mapsto \mathsf{false}\, \}
\end{array}
$$

This BE2B function will be used in the translation of JML assertions (like pre-conditions, post-conditions and invariants) to ordinary logic (of PVS).

# From JML- to ordinary predicates 3

The interpretation of the specification:

```
public static int firstElement (int[] array)
   /*@ normal_behavior
     @   requires : array.length > 0;
     @    ensures : true;
     @*/
   { return array[0]; }
```

will thus be the Hoare triple:

$$\Big[\,\mathsf{BE2B}\cdot[\![\,\mathsf{Inv}\ \&\&\ \texttt{array.length > 0}\,]\!]\,\Big]\ \texttt{firstElement}\ \Big[\,\mathsf{BE2B}\cdot[\![\,\mathsf{Inv}\,]\!]\,\Big]$$

where Inv is the class invariant.

Note: in case `array == null`, the specification is

- trivial to prove,
- but impossible to use.

# Logical consequences 1

With the "operational" semantics we use for the JML logic, there will be a difference between the assertions:

```
array != null &&
array.length > 0
```

and

```
array.length > 0 &&
array != null
```

But this will be unproblematic for Java programmers (and PVS users).

This means that one has to be careful at various stages, e.g.

- when there are multiple invariants, their order matters.
- the order of a pre-/post-condition and an invariant matters.

  We use: `Inv && Pre` and `Inv && Post`, because invariants usually express general safety properties (e.g. that certain references are not null).
- ...

## General Rule

Avoid possible exceptions in assertions !

## But this may be painful

Instead of

```
a[2] == b[3];
```

one should write

```
a != null && a.length > 2 &&
b != null && b.length > 3 &&
a[2] == a[3];
```

Again: our semantics makes it impossible to use such specifications.

(but trivial to prove when the exception occurs in the invariant or pre-condition)

## Ultimate Goal

Verification of JavaCard applets, on the basis of a verified specification of the JavaCard API.

## Current Goal

Specification of the JavaCard API, and verification of the current reference implementation (version 2.1).

The specification should be verified again for each new version of the library.

## Current subgoal

"Lightweight" specification (and verification) of the API, giving only termination, modification and invariant clauses, but no functionality.

Two examples, of native methods: `arrayCompare` and `arrayCopy` from `Util.java`

## arrayCompare in JML

```
public static final native byte arrayCompare(byte[] src,  short srcOff,
                                              byte[] dest, short destOff,
                                              short length)
   throws NullPointerException, IndexOutOfBoundsException;
/*@ normal_behavior
  @   requires: src != null && dest != null &&
  @             srcOff >= 0 && destOff >= 0 && length >= 0 &&
  @             srcOff + length <= src.length &&
  @             destOff + length <= dest.length;
  @ modifiable: nothing;
  @    ensures: true;
  @*/
```

## arrayCopy in JML

```
public static final native byte arrayCopy(byte[] src,  short srcOff,
                                          byte[] dest, short destOff,
                                          short length)
   throws NullPointerException, IndexOutOfBoundsException, TransactionException;
/*@ behavior
  @   requires: src != null && dest != null &&
  @             srcOff >= 0 && destOff >= 0 && length >= 0 &&
  @             srcOff + length <= src.length &&
  @             destOff + length <= dest.length;
  @ modifiable: dest[*];
  @    ensures: true;
  @    signals: (TransactionException) true;
  @*/
```

Note:

- `behavior` denotes a partial Hoare formula (actually: two)
- We do not assume that the reference `dest` itself is modified (but this is not really clear).
- We could be more precise and say that only `dest[destOff]...dest[destOff+length-1]` are modified

The JavaCard API contains an Application Identifier class `AID`, which contains one field:

```
byte[] theAID;
```

plus several associated methods: `getBytes`, `equals` (twice), `partialEquals`, `RIDEquals`.

The first five bytes are reserved for the RID (National Registered Application provider identifier). Eleven further bytes may be used for identification.

Therefor we have as AID class invariant in JML:

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

This property is established by the AID constructor, and preserved by all AID methods (whether terminating normally or abruptly).

## getBytes from AID

```
public byte getBytes (byte[] dest, short offset)
   throws ArrayIndexOutOfBoundsException, NullPointerException,
          TransactionException // not in the original source
    /*@ behavior
      @     requires: dest != null &&
      @               offset >= 0 &&
      @               offset + theAID.length <= dest.length;
      @   modifiable: dest[*];
      @      ensures: true;
      @      signals: (TransactionException) true;
      @*/
   {
     Util.arrayCopy(theAID, (short)0, dest, offset, (short)theAID.length);
     return (byte)theAID.length;
   }
```

## getBytes translation

Let

$$\mathsf{Pre} \;=\; \Big(\texttt{dest != null \&\& offset >= 0 \&\& offset + theAID.length <= dest.length}\Big)$$

$$\mathsf{Inv} \;=\; \Big(\texttt{theAID != null \&\& 5 <= theAID.length \&\& theAID.length <= 16}\Big)$$

$$\mathsf{Mod} \;=\; \Big(\cdots\Big) \quad // \quad \text{Interpretation of modifies clause: all memory cells remain the same, except the ones referred to.}$$

The LOOP tool translates the JML spec of `getBytes` into two partial Hoare triples in PVS:

$$\Big\{\,\mathsf{BE2B}\cdot[\![\,\texttt{Inv \&\& Pre}\,]\!]\,\Big\}\;\texttt{getBytes}\;\Big\{\,\mathsf{BE2B}\cdot[\![\,\mathsf{Inv}\,]\!]\wedge\mathsf{Mod}\,\Big\}$$

$$\Big\{\,\mathsf{BE2B}\cdot[\![\,\texttt{Inv \&\& Pre}\,]\!]\,\Big\}\;\texttt{getBytes}\;\Big\{\mathsf{Excp}\Big(\mathsf{BE2B}\cdot[\![\,\mathsf{Inv}\,]\!]\wedge\mathsf{Mod},\mathsf{TransactionException}\Big)\Big\}$$

## Normal termination

Assume the pre-condition and the invariant hold, and that getBytes terminates normally. We have to show:

1. Only the elements of the parameter array `dest` may have changed. This follows from the specification of arrayCopy, which can be used because its pre-condition is satisfied.
2. The invariant holds in the post-state. Again this follows from the specification of arrayCopy. We have to be careful about aliasing: possibly `dest == theAID`, but in that case the invariant remains valid because only the contents of `dest` may change.

## Abrupt termination

Assume the pre-condition and the invariant hold, and that getBytes terminates abruptly. We have to show:

1. Only the elements of the parameter array `dest` may have changed.
2. The invariant still holds.

The proofs are as before, except that we first need to show that the abrupt termination is caused by `arrayCopy`. Then the relevant specification can be applied.

## Source code

- The group of Arnd Poetzsch-Heffter in Hagen (Germany) does related work, using an ordinary Hoare logic (on the program level, instead of semantic level).

## Byte code

- The group of Tobias Nipkow in Munich (Germany) works on meta-level formalisation in Isabelle of large parts of the Java(Card) Virtual Machine, and the verifier (and compiler!).
- At INRIA in France there are various active groups using Coq for formalisation of the JavaCard platform, and for reasoning about bytecode programs.
- Several smart card manufacturers (like Bull and Gemplus) use formal methods (often with B) in order to get the highest Common Criteria assessment.
- There are various other active groups in this area, e.g. at Kestrel in California.

Many of these efforts in Europe have been combined in a research proposal VerifiCard, submitted to the European Community for funding.

# Conclusions

- JavaCard is an ideal target for the (tool-supported) use of formal methods: size, importance, interest, openness of the industry.

- JML is an appropriate language for specifying the behaviour of Java(Card) classes and methods.

- The coalgebraic semantics of Java developed within the LOOP project extends smoothly to JML, and brings forward many subtle issues in the semantics of specification.

  (The LOOP translation of JML is far from complete.)

- The actual verification results sofar are modest, but the fact alone that problems with such complexity can now be managed is already a significant step forwards.

More information at the URL `http://www.cs.kun.nl/~bart/LOOP`