VerifiCard

A European Project for
Smart Card Verification[*]

Bart Jacobs, Hans Meijer and Erik Poll
Computing Science Institute
University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
{Bart.Jacobs,Hans.Meijer,erikpoll}@cs.kun.nl
http://www.verificard.org

February 13, 2001

Abstract

The next generation of smart cards will be used for services where security is a key issue. Reliability and trust are necessary for a large scale adoption and success of these smart cards. New validation techniques are needed, based on well-defined mathematical models, using special tools for mathematically proving correctness, going well beyond testing. A EU-funded consortium 'VerifiCard' of 5 academic and 2 industrial partners, coordinated by Nijmegen, will work on the correctness of crucial components of the chosen (JavaCard) platform and of individual applications. This brief note will give an introduction to smart cards and their importance in computer science today. Additionally, it will give an impression of the work done in Nijmegen on these topics.

1 Introduction

Smart cards are small devices carrying a computer chip. They come in various shapes: plastic cards in the form of credit cards (used as, for instance, electronic purses) or SIM-cards in mobile telephones.

The next generation of smart cards will be used for services where security is a key issue, like authenticated access to computer networks, e-commerce, m-commerce, high value wire-less (GSM- or UMTS-based) services, loyalty programs and digital signing. The correct functioning of these cards must be absolutely guaranteed. Potentially malicious application programs must be identified. A virus on a smart card is a nightmare scenario. Therefore, new validation techniques are needed, based on well-defined mathematical models, using special tools (theorem provers and model checkers) for mathematically proving correctness, going well beyond testing.

[*]From: Newsletter 5 of the Dutch Association for Theoretical Computer Science (NVTI), 2001.

This offers both a challenge and an opportunity to the field of formal methods in general, and of JAVA-program verification in particular, as the leading smart card platform – JAVACARD – is based on JAVA. The challenge is to show that formal verification techniques are applicable to practical programs, in a real-life context, and thus may develop from an academic discipline into an industrially relevant field. The opportunity follows from the fact that smart cards are still small enough to bring their software within the reach of modern verification tools.

This is a very important point. The formalization, specification and verification of the JAVACARD platform is certainly not a trivial task, yet the platform is small enough to make a *complete* formal treatment feasible. Moreover, since smart cards are mass-produced and open to any kind of attack, the smart card industry as well as the card issuers have a profound interest in an error-free and secure implementation of the platform and its applets. This makes JAVACARD a unique chance for formal methods to prove their importance for the software industry. For a comprehensive overview of applications of formal methods to JAVACARD and smart cards, see [Har00].

2 JavaCard

Smart cards contain a microprocessor chip which can execute small application programs called *applets*. The cards that are in use today feature a single applet, written in machine code, which is 'burnt' in ROM and therefore fixed forever. In contrast, the new generation of smart cards can hold several applets in their memory at the same time ("multi-application"), and new applets can be downloaded after the cards have been issued ("post-issuance"), allowing services to be updated or new services to be added without replacing the card. Moreover, applets are not written in machine code specific to the particular chip used, but are written in a high-level language which is compiled to byte code that is interpreted by a virtual machine on the smart card.

The amount of memory available on smart cards is rather restricted, typically in the order of 10 kilobytes. It consists of a ROM containing a run-time environment, persistent updateable memory (such as EEPROM) for information which has to be preserved when power is removed, and transient memory for temporary workspace.

A card is used by inserting it in (or bringing it sufficiently close to) a smart card reader. This smart card reader will generally be an input/output device of some computer system, which communicates with the card's system by exchanging byte sequences called Application Protocol Data Units (APDUs). An APDU may contain an order to select a certain applet, or an order for the currently selected applet. The card responds with APDUs containing results, error messages, etc. This and other aspects of smart cards are standardized in ISO 7816.

Applets are written in a high level language and are designed to run on a standardized open platform. The applets for smart cards can be programmed in a language JAVACARD, which is a simplified version of the popular language JAVA. JAVACARD, like JAVA, is owned by Sun, but is freely available and described in open standards.

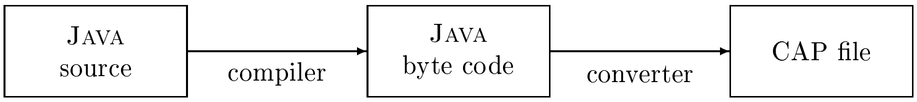
There are currently three standards for smart cards: JAVACARD[1], MultOS, and Windows for Smart Cards. JAVACARD is the most open standard, with its specifications publicly and freely available. All currently operational implementations of multi-application cards use the JAVACARD platform, and all major players in the smart card market are producing JAVACARD-based products. MultOS is slowly moving in the direction of JAVACARD. Windows for Smart Cards is developed by Microsoft. Altogether, the choice for JAVACARD seems obvious.

The JAVACARD language is a subset of JAVA developed with the limited memory and computation power of smart cards in mind. For instance, it only supports the types `byte`, `short`, and optionally `int` with their basic operations, but not `double` and `float`, and only allows one-dimensional arrays.

[1] See the URL of the JAVACARD Forum `http://www.javacardforum.org`

It supports most control flow constructs, as well as exception throwing and catching. However, its Application Programming Interface (API) is quite small, see below.

The source code of a JavaCard applet is translated into byte code by a standard Java compiler. The resulting byte code is checked by a *byte code verifier* to verify certain properties related to e.g. typing, and to see that the applet only uses features supported by JavaCard. Since Java byte code contains redundant information (which for instance enables full reverse-compilation), and JavaCard applets should be as small as possible, the byte code is further converted by a *cap converter* into a Converted Applet (CAP-file), which is the form loaded and installed on the card.



An applet's CAP file is interpreted by the JavaCard Virtual Machine (JCVM). The JCVM resides in the card's ROM, together with the JavaCard Run-time Environment (JCRE), which is an implementation of the JavaCard API, and the applets which are loaded when the card is manufactured. One of these applets may be designed to load and install *post-issuance* applets into persistent memory.

Applets are separated from each other by a *firewall* mechanism. No object created by one applet can be accessed by a different applet, unless such access is explicitly requested and granted, via a so-called *shareable object*.

Whenever the card is powered-up in a card reader it is reset to a consistent state. In order to preserve consistency, all updates of persistent memory are subject to a `begin/commit/abort`-transaction mechanism.

Altogether, the JavaCard API currently comprises 18 classes for exceptions, 16 interfaces for working with cryptographic keys and only 10 fundamental classes. Among the latter are a class `APDU` for communication with the card reader, classes `PIN` and `OwnerPIN` for working with PIN-codes, an abstract class `Applet` for applets, and a class `JCSystem`, which contains methods for transaction handling and for handling requests for object sharing between applets maintaining the applet firewall.

For general background information on smart cards, see [HNSS00], and for more specific information on JavaCard, see [Che00].

3 Formal methods for smart cards in VerifiCard

The VerifiCard project will work on formalization of the JavaCard platform (mainly: what is on the card) and of applets, in several different ways.

Platform specification and verification providing formal descriptions of the full JavaCard smart card platform, starting from the open JavaCard standards. This formalization includes that of the JavaCard language, virtual machine and run-time environment (API) and is used to develop and formally establish the soundness of several components of the JavaCard platform, in particular the byte code verifier and cap converter, and possibly also the compiler. It is also needed as a basis for techniques of applet verification.

Since the existing informal specification of JavaCard is partly given at source code level and partly at byte code level, the platform specification and verification will be done at both levels.

At the source code level, a formal operational semantics of the JavaCard language and API will be provided, for proving the compiler correct. As an axiomatic semantics (Hoare logic) is better suited to proving logical (safety) properties of applets using theorem provers, such a semantics is provided as well. The soundness and completeness of the axiomatic semantics w.r.t. the operational one will be proved.

At the byte code level, a formalization of the JCVM will be provided, and in doing so, a formal operational semantics of JAVA byte code. This formalization will be used for a verification of the compiler, and the construction of a certified byte code verifier and cap converter.

Applet specification and verification developing methods and tools for the validation of aspects of applets, in particular security aspects. The applets run on the JAVACARD platform, and therefore these methods need to be based on the platform formalisation.

Different styles of specification will be explored, namely logic methods using Hoare logic and algorithmic methods using temporal logic, in order to investigate the types of problems for which they are best suited. The case studies provided by the industrial partners will drive this exploration (instead of looking for problems which are well solved by a chosen method).

The algorithmic methods will be based on an operational semantics at byte code level. A JAVACARD Temporal Logic Specification Language will be developed which should take care of typical aspects of JAVACARD like object sharing and firewalls, transactions, transient objects, *etc.* Three techniques for the verification of such aspects will be considered: compositional proof techniques (allowing to specify and verify applets individually), abstract interpretation and model checking.

The logic methods will be based on an axiomatic semantics at source code level. A JAVACARD Interface Specification Language will be developed which is similar to a subset of JML described below. Three approaches to verification are investigated: verification at the syntactic level and at the semantic level, the latter with and without using a representation of the global object store. For verification at the semantic level applets are translated into the logic of some theorem prover, using the LOOP-tool described below.

Applications applying and testing the techniques developed in the other parts, in particular using realistic case studies supplied by industrial partners and members of the End-User Panel.

In particular a banking case study and a mobile communication (GSM) case study will be developed which will include examples of hostile applets to be detected as such by the methods and tools developed earlier. Also, a set of security properties will be provided which are to be verified by these methods and tools. As a further application, the API will be verified using the specification developed earlier.

In the project 5 academic and 2 industrial partners participate. The academic partners are the University of Nijmegen (the project coordinator), INRIA in France, the Technical University of Munich, the University of Hagen in Germany and SICS, the Swedish Institute of Computer Science. The industrial partners are the French companies Gemplus and Bull, who are both deeply involved in the development of (JAVACARD) smart cards.

The project also has an End User Panel representing different industries with an interest in smart card security. The End User Panel currently consists of TNO (The Netherlands), Deutsche Telekom, France Telecom, Setec Oy (Finland), Trusted Logic (France), IBM Zurich, INTEGRI (Belgium) and Ericsson (Sweden).

At source code level, Munich will work on the operational and axiomatic semantics of JAVACARD. Nijmegen and Hagen will develop a complete formal specification of the JAVACARD API, which ideally should become a standard reference. All this will form the basis for the specification and verification of JAVACARD applets at source code level using theorem provers by Nijmegen, Hagen, and INRIA. Here Nijmegen will use the LOOP-tool – discussed in more detail below –, Hagen will use the Jive tool developed there, and INRIA will use Coq.

At byte code level, INRIA, Munich, and Gemplus will work on formalisation of the JavaCard virtual machine and a certified byte code verifier. INRIA will also develop a certified cap-converter, and Munich will start work on a certified compiler. The formalisation of the virtual machine will be used by INRIA and SICS for verification of JavaCard applets at byte code level using modelchecking, abstract interpretation, and compositional proof techniques.

4 The Java Modelling Language (JML)

Since new applets can be downloaded to the new generation of cards, card issuers are very worried about controlling which applets are allowed on their cards. They can use digital signatures to ensure that only applets that they signed can be installed. But this leaves them with the problem that they want to know for sure that their applets behave correctly. This leads to old-fashioned program verification, which, these days, is done with modern (verification) tools and formal specification languages. This is the topic that Nijmegen (and also Hagen and INRIA) will work on within the VerifiCard consortium.

The LOOP-group in Nijmegen (see [Loop]) uses the behavioural specification language for Java called JML [LBR99b, LBR99a]. JML is designed primarily by Gary Leavens in cooperation with Compaq SRC and Nijmegen. We give a brief account of its features without going into any detail.

JML is designed to specify the *behaviour* of Java classes and interfaces and their objects. The basic idea is that pre- and postconditions be specified for each method of a class, and an invariant for the class as a whole, which describes properties of the class' fields, but may be considered as an additional pre- and postcondition for the class' methods.

One may specify several 'behaviours' for a method. Each behaviour has a precondition given as a `requires`-clause and a postcondition given as an `ensures`-clause. A behaviour may have additional clauses, like a `modifiable`-clause which specifies (non-local) variables which may be changed in that behaviour, or a `signals`-clause with a condition, which indicates that the behaviour may result in throwing an exception when that condition holds. A behaviour may be qualified as 'normal', in which case it should return normally, or 'exceptional', in which case it should signal an exception. JML knows many other qualifying modifiers.

The invariants and pre- and postconditions are written as (Boolean) expressions in standard Java-syntax, extended with various constructs like `forall` and `exists`. The reason for choosing standard syntax is to encourage Java-programmers to specify their code as they write it. Extensions include quantifiers and set comprehension, logical implication, a pseudo-variable `\result` indicating a method's returned value, and more such pseudo-variables and pseudo-operators like `\old`: the pre-value of a modifiable variable v may be referred to in postconditions as `\old(`v`)`.

A specification may use *model fields*, i.e. specification-only variables, e.g. to capture implicit variables whose values could be inferred from explicit Java variables. The initial value of such model fields may be specified. A specification may also directly call methods of 'pure' classes, i.e. without side effects. This allows for the abstraction of (mathematical) details in a Java setting.

Specifications are written in separate files or embedded in Java code as special comments.

As an example, a JML specification of a method `ArrayCopy` of the JavaCard API class `Util` is given in Fig. 1. The intended behaviour of `arrayCopy(src, srcOff, dest, destOff, length)` is to copy the items from `src[srcOff..srcOff+length-1]` to `dest[destOff..destOff+length-1]`.

A subtle point is the need for `\old(...)` in the `ensures` clause, i.e. the postcondition, of the `arrayCopy` method. This is needed to correctly specify the behaviour in the case that aliasing occurs, i.e. the case that `src == dest`. We have to refer to the original entries of the `src` array in the postcondition. The informal (javadoc) specifications of the JavaCard API explicitly state

```
/*@ 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;
  @*/
public static final native short arrayCopy(byte[] src,
                                           short srcOff,
                                           byte[] dest,
                                           short destOff,
                                           short length)
throws ArrayIndexOutOfBoundsException,
       NullPointerException,
       TransactionException;
```

Figure 1: JML specification for `arrayCopy` from `Util`

that this is what happens if `src` and `dest` are aliases.

The `TransactionException` that may be thrown by `arrayCopy` is typical for JavaCard, and may arise because of the limited available resources on smart cards. It may occur when an overflow arises in a special transaction buffer that is used to enable rollback of operations in case of failure (*e.g.* when the card is prematurely removed from the card reader.) Further examples of JML specifications for classes from the JavaCard API are discussed in [PBJ00, PBJ01].

Specifications as those in Fig. 1 provide a basis to formally prove that a given method implementation (*e.g.* in an applet, possibly involving an `arrayCopy`) satisfies its specification. Especially the `modifiable` clauses are useful to control the side-effects that applets can have (and thus the possible damage that they can do).

5 The LOOP-tool

The LOOP-tool (where LOOP stands for Logic of Object Oriented Programming) is developed in Nijmegen, partly in cooperation with the Technical University of Dresden. It translates JML-programs (i.e., Java programs annotated with JML-specifications as outlined above) into a description of their semantics in the higher-order logic used by the theorem provers PVS and Isabelle/HOL, see [BJ00]. In fact, a Java-class is modelled as a co-algebra [JR97, Rei95, Jac99] mapping a state into a sum of products of Java types and the state space. The theorem provers PVS or Isabelle/HOL can then be used that Java-programs meet their JML-specifications. This is a non-trivial, highly interactive activity for which a special tailor-made Hoare logic has been developed [JP00]. The first steps in the verification of JML-annotated Java-programs using the LOOP tool are described in [BJP01].

6 Summary

The JavaCard platform offers a unique opportunity for formal methods to show their value for real-life software development: the platform is non-trivial but not too large, and an error-free and secure implementation is of vital importance not only for the smart card industry and card issuers, but also for society at large.

The VerifiCard project aims to provide methods and tools for the complete specification and verification of applets written for the JavaCard platform used in next-generation smart cards. To this end the project will develop

- a formalization of
  - the JavaCard language at source level, by an operational semantics and an axiomatics semantics, and of the JavaCard Application Programming Interface (API)
  - the JavaCard Virtual Machine, giving an operational semantics of the JavaCard byte code level
- certified tools for developping JavaCard programs, including
  - a byte code verifier
  - a cap converter
  - a Java compiler
- methods and tools for the specification and verification of applets, at byte code level and source code level, using theorem proving, abstract interpretation, and model checking.

These formalizations, specifications and verification methods will be evaluated using non-trivial characteristic case studies (and the reference implementation of the JavaCard API).

References

[BJ00] 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.

[BJP01] J. van den Berg, B. Jacobs, and E. Poll. Formal Specification and Verification of JavaCard's Application Identifier Class. In I. Attali and T. Jensen, editors, *Proceedings of the JavaCard Workshop*, LNCS. Springer, 2001.

[Che00] Z. Chen. *Java Card Technology for Smart Cards.* The Java Series. Addison-Wesley, 2000.

[HNSS00] U. Hansmann, M.S. Nicklous, T. Schäck, and F. Seliger. *Smart Card Application Development Using Java.* Springer, 2000.

[Har00] P. Hartel. Formalising Java safety – An overview. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000)*, pages 114–134. Kluwer Acad. Publ., 2000.

[Jac99] B. Jacobs. Coalgebras in specification and verification for object-oriented languages. Newsletter 3 of the Dutch Association for Theoretical Computer Science (NVTI), 1999.

[JP00] 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.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[LBR99a] 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.

[LBR99b] 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.

[Loop] Loop Project. `http://www.cs.kun.nl/~bart/LOOP/`.

[PBJ00] 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.

[PBJ01] 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.

[Rei95] H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.