Formalization in Coq of the Java Card Virtual Machine

Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard Serpette, Simão Sousa, and Shen-Wei Yu

Projet Oasis
INRIA Sophia-Antipolis
France
e-mail: `FirstName.LastName@sophia.inria.fr`

1 Introduction

Java has quickly become a standard for Internet and, through Java Card, for smartcard programming, putting security issues at stake. Yet recent research has unveiled several problems in the Java Card security model, most notably with object sharing and the associated mechanism of shareable interfaces, see e.g. [10, 17]—there are further security breaches related to some Java features not present in Java Card but we are not concerned with them here. It is therefore fundamental to develop environments to verify the security of the Java Card platform and of Java Card programs. Thus far Java/Java Card security has been studied at two main levels:

1. platform level: here the goal is to prove safety properties of the language, in particular type safety and properties related to memory management. This first level may be instrumented with proof assistants, which are well-suited to formalization and logical reasoning;
2. application level: here the goal is to prove that a specific program obeys a specific security policy, such as the sandbox security policy or information-flow based security policies. This second level may be instrumented with a variety of tools, such as model-checkers, which allow to perform automatically a symbolic evaluation of programs to check whether a given program adheres to a given security policy.

The long-term goal of our work, part of which is being carried in the context of the *Action de Recherche Coopérative* S-Java (see `http://www-sop.inria.fr/oasis/SJava`), is to provide an environment that integrates both layers and allows to prove security properties for Java Card programs. In the present article, we set up the foundations for such an environment by providing:

1. an executable formal specification in Coq of a substantial fragment of the JCVM and JCRE. The specification is written in functional style and formalizes one step execution of the JCVM by a function `exec_jcvm` of type

   ```
   jcvm_program -> jcvm_state -> (Exc jcvm_state)
   ```

   where `jcvm_state` is the type of JCVM states (see Section 2), and `Exc` is the Coq representation of the lift monad. Our specification is rather precise: in particular, it implements the mechanisms of firewalls and shareable interfaces upon which Java Card security crucially relies;
2. a tool, written in Java, that translates CAP files into their Coq representation. In order to achieve a compact and readable representation of Java Card programs, the CAP tool performs various program transformations, including linking and index manipulation, and may be viewed as a fair part of a bytecode verifier.

The remaining of the abstract is organized as follows: in Section 2, we present our formalization of Java Card programs and of the JCVM and JCRE. In Section 3, we present the CAP tool that translates CAP files into their Coq formalizations. We conclude in Section 4 with related work and perspectives for future research.

2 Formalization of the JCVM

2.1 Java Card programs

Java Card programs are formalized as a record of lists of classes, methods and interfaces:

```
Record jcvm_program : Set := {
       classes    : (list Class);
       methods    : (list Method);
       interfaces : (list Interface)
       }.
```

The `Class` and `Interface` types are records that contain several fields not described in this abstract. The bytecode instructions to be executed by JCVM are encapsulated in the `Method` type:

```
Record Method : Set := {
       nargs      : nat;                (* Number of arguments of the method. *)
       local      : nat;                (* Number of local variables. *)
       owner      : nat;                (* Class index. *)
       bytecode   : (list Instruction)  (* All the instructions to be executed. *)
       }.
```

Here `Instruction` is an inductive type specifying the mnemonic and the format of all the instructions as sketched below:

```
Inductive Instruction : Set :=
          ... |
          invokevirtual   : nat -> nat -> Instruction |
          ...
```

The `invokevirtual` instruction is formalized as a constructor taking as arguments two naturals that respectively represent the number of arguments and the position of the invoked virtual function in the method list of the class of the object under consideration.

2.2 Memory model

The JCVM memory consists of two main elements : the `heap` and the `stack`. The `heap` involves all the objects that an application may use. It is formalized in Coq using a list of objects. The latter are either class instances, described using a class reference and a data area (a list of values), or arrays, constructed from the type of their elements and another data area. The type of values is the cartesian product `type*Z` where `type` represents the data types supported in Java Card (`Void`, `Boolean`, `Byte`, `Short`, `Int`, `Ref`) and `Z` is the type of integers.

The `stack` contains all the running methods and it is represented as a list of frames. Frames are described in Coq as a record type featuring an operand stack, a list of local variables, a reference to the current class, the method location, the current context (the type `Package` contains information that allows to identify a context) and a program counter that points to the next instruction to be executed:

```
Record frame : Set := {
       opstack       : (list valu);
       locvars       : (list valu);
       current_class : nat;
       method_loc    : nat;
       context_ref   : Package;
       p_count       : nat
       }.
```

2.3 One-step execution

We have formalized in Coq one-step execution function of the JCVM as a term `exec_jcvm` of type

```
jcvm_program -> jcvm_state -> (Exc jcvm_state)
```

where `jcvm_state` is the cartesian product of the heap, the stack and the exception that may be returned. The term `exec_jcvm` is built from auxiliary functions that describe the behavior of each instruction. Our current formalization treats most instructions of the JCVM, including `checkcast`, `athrow`, `instanceof`, `invokeiterface`, `aastore`, `new`, `getfield`. In this abstract, we focus on the formalization of the `invokevirtual` instruction, which performs the invocation of an instance method.

The `invokevirtual` instruction performs the invocation of an instance method. Its semantics is captured by the Coq function `INVOKEVIRTUAL`. The function returns an element of type `(Exc jcvm_state)` and is described below.

```
Definition INVOKEVIRTUAL := [nargs,idx:nat][h:frame][hp:heap][s:stack][cap:jcvm_program]

(* The number of arguments must be greater than zero *)
 Cases nargs of
  O     => Error | (S _) =>

    (* The object reference which is of type valu is the nargth element of the operand stack.
       An element of valu is a pair (t,v) where t is the type of the value v.  *)
    Cases (Nth_func (opstack h) nargs) of (value x) =>
      Cases x of (tx,vx) =>

         (* The nargth element must be a reference. *)
         Case tx of Ref =>

           (* This function raises a Null Pointer exception (Value NullPointer)
              if the value vx is null, otherwise returns error. *)
           Cases (test_nullpointer vx) of error=>

               (* The referenced object is extracted from the heap.
                  An object in the heap is either an instance or an array.  *)
                          Cases (Nth_func hp (absolu vx)) of
                             (value nhp) =>  Cases nhp of

                                               (* Tests if this object is an instance. *)
                                               (Instance ti) =>
                                                 Cases ti of (Build_type_instance c _ _ _ _ _ _) =>

                                                   (* The current frame is updated and the new
                                                   frame is set up and pushed into the stack. *)
                                                   (res_invoke c nargs idx h hp s cap nhp)
                                                 end |
                                              (Array ta)   => ...
```

The `new_frame_invokevirtual` function, called by `res_invoke`, returns an element of type `jcvm_state` and, if needed, updates the stack frame.

```
Definition new_frame_invokevirtual:=
[cref:nat][idx:nat][l,l':(list valu)][h:frame][hp:heap][s:stack][nhp:obj]
  Cases (jcre_invoke h nhp) of

     (* If an exception is raised then an error state is returned
        in which u is the thrown exception. *)
     (value u) =>((Value u),(hp,s))

     (* Else the current frame is updated, a new frame is set up,
        pushed into the stack and becomes the new current frame.
        No exception is thrown. *)
      error    =>((Error), (hp,
                           (cons (Build_frame (nil valu)  (rev l) cref idx
```

```
                                      (get_owner_context nhp) (* The currently active context
                                                                 is the object owner's context. *)
                                      (1))                    (* The program counter points
                                                                 to the first instruction. *)
                                (cons (update_current_frame l' h)
                                      (tail s)))))
  end.
```

The JCVM may throw an exception under certain conditions. For example:

- a `Nullpointer exception` is thrown if the object reference extracted from the operand stack is null;
- a `Security exception` is thrown if the Java Card security policy is violated.

The following example is part of the `jcre_invoke` Coq function which performs security checks upon instances (recall that objects are either instances or arrays).

```
Definition jcre_invoke := [h:frame][the_object:obj]

...
(* Tests if the referenced object is an instance. *)
Cases the_object of (Instance ti) =>

   (* Tests if the currently active context is the object owner context. *)
   Cases (eqb_AID (AID_pi (context_ref h)) (AID_pi (owner_i ti))) of
     true  => Error |
     false =>

        (* Tests if this object is an Entry Point. *)
        Cases (eqb (ptE ti) is_ptE) of true  => Error |
           false =>

              (* Tests if the currently active context is the JCRE context. *)
               Cases (eqb_AID (AID_pi  (context_ref h)) jcre_AID) of true  => Error |

                 (* A Security Exception is thrown. *)
                 false => (Value Security)
               end ...
```

2.4 Discussion

Our formalization uses a "functional style" of specification. In our opinion, such a style offers several advantages over the usual "logic programming" style of formalization using inductive relations. In particular, a functional style specification

1. is close to an implementation and may be tested for increasing confidence in the formalization;
2. is well suited to verify program properties that require an exhaustive check of execution traces, including security properties like information flow. In particular, functional style specifications are easily amenable to model-checking;
3. is re-usable in other proof-assistants, as it mostly relies on standard typing constructs such as first-order inductive types and record types, and do not make use of dependent types—in fact, we only make use of a non-standard feature of Coq, namely implicit coercions, to simulate subtyping on inductive types in the representation of Java Card programs.

3 The CAP tool

In order to test our formalizations, we have developed a CAP tool that transforms CAP files into their Coq representations. To this end, the CAP tool performs the following operations:

1. *Linking.* During this stage, the constant pools (reference tables) of different packages are fully resolved (for external references) and thus can finally be removed. This stage also involves transforming the flat format of the CAP files into a more realistic tree structure. For instance, bytes indexes in the `Method component` are translated to pointers to `method` objects;
2. *Indexes transformation.* The JCVM specification and the CAP file format are word oriented: for example the stack is described as a short vector and thus `int` local variables must take two consecutive indexes. In contrast, our Coq specification is value oriented: the stack is a list of `valu` and a method contains a list of `Instruction`. Therefore, the CAP tool must also make a deep transformation of the byte code:
   - indexes in the byte code (for example for the `goto` instruction) are resolved as `Instruction` indexes;
   - indexes in the stack (for example for the `aload` instruction) are resolved as `valu` indexes.

   Such a transformation can be viewed as a fair part of a byte code verifier.

4 Conclusion

We have reported about a precise formalization of a substantial part of the JCVM in Coq. This work is part of a larger effort to build up a toolbox for proving security properties of Java Card programs and many facets of the project still remain to be addressed.

4.1 Future work

Our most immediate objective is to complete the formalization of the JCVM and JCRE. In parallel, we intend to develop tools for verifying security properties that are not enforced by the JCVM security mechanisms. For example, the Java Card language does not make any provision to prevent information leakage through transitive information flow; in order to verify the absence of such information leakage, we plan to use [23] to set up a connection between our formalization of the JCVM and model-checkers.

In the longer term, we are also interested in integrating abstract interpretations to our environment. Ideally, these abstract interpretations should be certified within Coq itself—preliminary work in this direction may already be found in [16].

4.2 Related work

Due to space constraints, we mostly focus on projects that make use of specification and/or verification tools for Java or Java Card. For the sake of clarity, we distinguish between platform-oriented projects and application-oriented projects.

Platform-oriented projects The proof assistant Coq has been used in a number of case studies on the JVM. For example, Bertot [2] has formalized the type system of Freund and Mitchell [9], and Segouat [21] has formalized the correctness of the converter from bytecode to CAP files following Denney and Jensen [6].

Yet the most impressive work to date is that of the Bali team at Munich, see e.g. [18], who formalized in Isabelle/HOL a large body of the Java platform, including:

1. the type system and the operational semantics of both the source language, with a proof of type-safety at both levels;
2. the compiler and an abstract bytecode verifier, with their proof of correctness.

In addition, there have been a number of similar efforts by Syme [22], who formalized the operational semantics of Java and machine-checked the proof of type soundness in DECLARE, by Lanet and Requet [15] who formalized most of the JCVM in B, and by Cohen [4] who formalized the so-called Defensive JVM in ACL2.

Application-oriented projects The LOOP project see e.g. [13] has developed a tool to compile Java classes into PVS or Isabelle theories that form the basis for actual verifications. Other projects, such as the Java Path Finder [12] and Bandera [5], emphasize the use of algorithmic techniques for verifying properties of Java programs. For example, the Bandera toolset combines several program analyses/program transformation techniques, including slicing and partial evaluation, to extract from Java source code compact finite-state models to be submitted to model-checkers.

Specification projects A number of teams have developed formal specifications of Java and its variants: these include executable specifications as in [11] or precise pen-and-pencil specifications as in [3, 1, 8]. There is also a variety of works that focus on specific aspects of Java, see e.g. [7, 14, 20], and Java Card, see e.g. [17, 19].

Acknowledgements Thanks to Henrik Nilsson for numerous discussions on the JCVM.

References

1. P. Bertelsen. Semantics of Java Byte Code. Master's thesis, Department of Computer Science, Royal Veterinary and Agricultural University of Copenhagen, 1997.
2. Y. Bertot. Formalizing in Coq a type system for object initialization in the Java bytecode language. Talk at the SJava meeting, Sophia-Antipolis, March 2000.
3. E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 353–404. Springer-Verlag, 1999.
4. R. M. Cohen. Defensive Java Virtual Machine Specification Version 0.5. Manuscript, 1997.
5. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of ICSE'2000*, 2000.
6. E. Denney and T. Jensen. Correctness of Java Card method lookup via logical relations. In G. Smolka, editor, *Proceedings of ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 104–118. Springer-Verlag, 2000.
7. S. Drossopoulou. Towards an abstract model of Java dynamic linking and verification. Manuscript, 2000.
8. S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and Proving Type Soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 41–82. Springer-Verlag, 1999.
9. S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. In *Proceedings of OOPSLA '98*, volume 33(10) of *ACM SIGPLAN Notices*, pages 310–328. ACM Press, October 1998.
10. P. Girard. Which security policy for multiapplication smart cards? In *Proceedings of Usenix workshop on Smart Card Technology (Smartcard'99)*, 1999.
11. P. H. Hartel, M. J. Butler, and M. Levy. The Operational Semantics of a Java Secure Processor. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 313–352. Springer-Verlag, 1999.
12. K. Havelund. Java PathFinder—A Translator from Java to Promela. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proceedings of 5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*, page 152. Springer-Verlag, 1999.
13. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes. *ACM SIGPLAN Notices*, 33(10):329–340, October 1998.
14. T. Jensen, D. Le Métayer, and T. Thorn. Security and Dynamic Class Loading in Java: A Formalisation. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*, pages 4–15, 1998.
15. J.-L. Lanet and A. Requet. Formal Proof of Smart Card Applets Correctness. In *Proceedings of CARDIS'98*, 1998. To appear.
16. D. Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Rapport de DEA, Université Paris VII, 1998.
17. M. Montgomery and K. Krishna. Secure Object Sharing in Java Card. In *Proceedings of Usenix workshop on Smart Card Technology, (Smartcard'99)*, 1999.
18. T. Nipkow, D. von Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proceedings of the International Summer School Marktoberdorf 1999*, pages xxx–xxx. IOS Press, 2000. To appear.
19. M. Oestreicher and K. Krishna. Object Lifetimes in Java Card. In *Proceedings of Usenix workshop on Smart Card Technology, (Smartcard'99)*, 1999.
20. Z. Qian. A formal specification of Java[TM] virtual machine instructions for objects, methods and subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 271–312. Springer-Verlag, 1999.
21. G. Ségouat. Preuve en Coq d'une mise en oeuvre de Java Card. Master's thesis, University of Rennes 1, 1999.
22. D. Syme. Proving Java type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 83–118. Springer-Verlag, 1999.
23. K. N. Verma and J. Goubault-Larrecq. Reflecting BDDs in Coq. Technical Report RR3859, INRIA projet Coq, January 2000.