A Formal Executable Semantics of the JavaCard Platform

Gilles Barthe[1], Guillaume Dufay[1], Line Jakubiec[1,2], Bernard Serpette[1], and Simão Melo de Sousa[1,3]

[1] INRIA Sophia-Antipolis, France
{Gilles.Barthe,Guillaume.Dufay,Bernard.Serpette,Simao.Desousa}@inria.fr
[2] Université de Provence, Marseille, France jakubiec@lim.univ-mrs.fr
[3] Universidade da Beira Interior, Covilhã, Portugal

Abstract. We present a formal executable specification of two crucial JavaCard platform components, namely the Java Card Virtual Machine (JCVM) and the ByteCode Verifier (BCV). Moreover, we relate both components by giving a proof of correctness of the ByteCode Verifier. Both formalisations and proofs have been machined-checked using the proof assistant Coq.

1 Introduction

1.1 Background

JavaCard [17] is a popular programming language for multiple application smart cards. According to the JavaCard Forum [16], which involves key players in the field of smart cards, including smart card manufacturers and banks, the JavaCard language has two important features that make it the ideal choice for smart cards:

- JavaCard programs are written in a subset of Java, using the JavaCard APIs (Application Programming Interfaces). JavaCard developers can therefore benefit from the well-established Java technology;
- the JavaCard security model enables multiple applications to coexist on the same card and communicate securely, and in principle, enables new applications to be loaded on the card after its issuance.

Yet recent research has unveiled several problems in the JavaCard security model, most notably with object sharing. This has emphasised the necessity to develop environments for verifying the security of the JavaCard platform and of JavaCard programs. Thus far JavaCard security (and also Java security) has been studied mainly at two levels:

- platform level: here the goal is to prove safety properties of the language, in particular type safety and properties related to memory management;
- application level: here the goal is to prove that a specific program obeys a given property, and in particular that it satisfies a security policy, for example based on information flow.

D. Sands (Ed.): ESOP 2001, LNCS 2028, pp. 302–319, 2001.

Over the last few years, both fields have been the subject of intensive investigations, see Subsection 6.1. Despite impressive progress, much work remains to be done. In particular, there is no complete formalisation of the JavaCard platform as yet nor widely used tools to verify applets' properties. Besides, we do not know of any environment that supports verification both at platform and application levels.

1.2 Our Work

The main contributions reported here are (1) a formal executable specification of two crucial JavaCard 2.1. platform components, namely the Java Card Virtual Machine JCVM and the ByteCode Verifier BCV; (2) a machine-checked proof of correctness of the ByteCode Verifier. Both formalisations and proofs have been carried out in the proof assistant Coq [4]. The salient features of our formal specification are:

- *executability.* Our formal semantics (both of the virtual machine and of the verifier) may be executed on any JavaCard program (given a Coq implementation of the native methods used by the program) and its behaviour can be checked against reference implementations, in this case Sun's implementation of the JavaCard Virtual Machine. We view executability as a crucial asset for reliability and, in our opinion, a formal operational semantics for a (realistic) programming language must be executable;
- *completeness.* Our virtual machine is complete in the sense that it treats the whole set of JavaCard instructions and it considers all the important aspects of the platform, including the firewall mechanism around which JavaCard security is organised. Our ByteCode Verifier handles the whole set of instructions but (1) it does not treat object initialisation; (2) subroutines are treated in a somewhat restrictive way.
- *suitability for reasoning.* Our formalisation may be used to reason about the JavaCard platform itself, as shown here, but also to prove properties of JavaCard programs. In particular, our formalisation is well-suited to reason about security properties formulated as temporal-logic properties over execution traces [28].

Thus our development offers the most comprehensive to-date machine-checked account of the JavaCard platform and compares well to similar efforts carried out in the context of Java, see 6.1.

1.3 JavaCard vs. Java

JavaCard is an ideal language for formal verification, since it is a reasonably-sized language with industrial applications. As compared to Java, the JavaCard Virtual Machine (in its current version) lacks garbage collection, dynamic class loading and multi-threading. In contrast, the firewall mechanism is a complex feature that is proper to JavaCard.

1.4 Organisation of the Paper

The remaining of this paper is organised as follows: in Section 2, we describe our formalisation of JavaCard programs (after linking). In Section 3, we describe a small-step operational semantics of the JavaCard Virtual Machine, where each instruction is modelled as a state transformer. In Section 4, we derive from the virtual machine an abstract virtual machine that operates on types (instead of values) and prove its correctness. In Section 5, we use the abstract virtual machine to build a ByteCode Verifier and prove it correct. In Section 6, we conclude with related and future work.

1.5 A Primer on Coq

Coq [4] is a proof assistant based on the Calculus of Inductive Constructions. It combines a specification language (featuring inductive and record types) and a higher-order predicate logic (via the Curry-Howard isomorphism). All functions in Coq are required to be terminating. In order to enforce termination, recursive functions must be defined by structural recursion. Besides, all functions are required to be total. To handle partial functions, we use the lift monad which is introduced through the inductive type:

```
Inductive Exc[A:Set]:Set := value: A->(Exc A) | error: (Exc A)
```

Our specifications only make a limited use of dependent types—a salient feature of Coq. This design choice was motivated by portability; by not using dependent types in an essential way, our formalisations can be transposed easily to other proof assistants, including PVS and Isabelle.

We close this primer with some notation. We use `*` to denote cartesian product of two types, `(a,b)` to denote pairs, `[x:A] b` to denote a λ-abstraction, `(x:A) B` to denote a dependent function space. Finally, a record type `R` is represented as an inductive type with a single constructor `Build_R`. Selectors are functions (defined by case-analysis) so we write `l a` instead of the more standard `a.l`.

2 Representation of JavaCard Programs

JavaCard programs are nothing but Java programs satisfying additional constraints. They can be compiled on a class by class basis by a standard compiler, yielding a class file for each class being compiled. For the purpose of JavaCard, compilation is followed by a further transformation phase where a converter transforms the set of class files corresponding to a package into a single CAP file, provided the former are JavaCard compliant. Finally, CAP files are linked before execution (recall JavaCard does not support dynamic class loading); during this last phase, constant pools are resolved and eliminated. Our representation of programs is based on this last format.

2.1 Representation of Data Structures

The JavaCard Virtual Machine distinguishes between primitive types and reference types (for instances of arrays, classes and interfaces). We use a mutual inductive type to enforce the distinction. Formally, the type of primitive types is defined (in Coq) by:

```
Inductive type_prim : Set :=
  Byte          : type_prim |
  Short         : type_prim |
  Int           : type_prim |
  Boolean       : type_prim |
  Void          : type_prim |
  ReturnAddress : type_prim.
```

while the type of (JavaCard) types is defined by:

```
Mutual Inductive type : Set :=
  Prim : type_prim -> type |
  Ref  : type_ref -> type
with type_ref : Set :=
  Ref_array     : type -> type_ref |
  Ref_instance  : nat -> type_ref  |
  Ref_interface : nat -> type_ref.
```

In principle our representation of types allows to form arrays of arrays, which is not permitted in JavaCard. However our formalisation, in particular the implementation of `anewarray`, does not allow to form such a type. (It is also straightforward to modify our formalisation not to allow such types to be formed.)

2.2 Representation of Programs

A JavaCard program is simply represented by its interfaces, classes and methods:

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

Note that, by default, interfaces and classes of the `java.lang` package and instances of these classes are an integral part of our program. This includes in particular the class `Object`, the interface `Shareable`, and Exception classes.

The types `Interface`, `Class` and `Method` are themselves defined as record types. We briefly describe the structure of classes and methods below. Interfaces are described in the full version of this paper.

Classes. A class is described by its superclasses[1] (if any), its methods (including constructors and distinguishing between public methods and package methods), the interfaces it implements, its class variables and, in the case of Java Card, its owning package. For execution purposes, we also need to keep track of the index of the class. Formally, we use the following structure to represent classes:

```
Record Class : Set := {
          (* List of all super classes of this class *)
  super           : (list class_idx);
          (* List of public methods *)
  public_methods : (list Method);
          (* List of package methods *)
  package_methods : (list Method);
          (* List of implemented interfaces *)
          (* For each interface we provide the list of methods *)
          (*  implementing the  interface's methods. Methods   *)
          (*  are tagged with their visibility.                 *)
  int_methods     : (list interf_idx*(list vis_method_idx)));
          (* List of types of class variables *)
  class_var       : (list type);
          (* Identification of the owner package*)
  package         : Package;
          (* Index of class *)
  class_id        : class_idx
}.
```

where `class_idx` and `interf_idx` are the types of indexes for classes and interfaces respectively and `vis_method_idx` is the inductive (sum) type:

```
Inductive vis_method_idx : Set :=
             pub_method_idx: method_idx -> vis_method_idx
          |  pac_method_idx: method_idx -> vis_method_idx.
```

where `method_idx` is the type of method indexes (the constructors are used to flag methods' visibility).

Our representation does not take into account the maximum depth of the operand stack during execution of the method. It is a simple matter to include this information but, during execution, we would need to perform many checks.

Methods. A method is characterised by its status (whether it is static or not), its signature (against which one can type-check its arguments upon invocation),

[1] Our description of a class `c` refers to all the classes from which `c` inherits, i.e. to which `c` is related by the transitive closure of the superclass relation. For execution purposes, these classes are gathered into a list. Our convention is that the immediate superclass of `c` appears first in the list. This encoding is chosen to avoid defining functions by well-founded recursion over the subclass relation.

its number of local variables (for initialising its execution context), its exception handlers, its list of instructions to be executed and finally the indexes of the method and of its owning class. Formally, we use the following structure to represent methods:

```
Record Method : Set := {
  (* Indicates whether a method is static or not *)
  is_static : bool;
  (* Signature of the method, pair of domain / codomain *)
  signature    : ((list type)*type);
  (* Number of local variables. *)
  local        : nat;
  (* List of exception handlers *)
  handler_list : (list handler_type);
  (* List of all instructions to be executed. *)
  bytecode     : (list Instruction);
  (* Index of the method in program*)
  method_id    : method_idx;
  (* Index of the owning class *)
  owner        : class_idx
}.
```

where the type `handler_type` collects the information required to define the best handler for a given program counter and exception. Formally, we use the following structure to represent handler types:

```
Definition handler_type :=
               (bytecode_idx*bytecode_idx*class_idx*bytecode_idx).
```

The first two elements define the range at which the exception handler is active. The third element defines the class of exceptions that the handler is meant to catch, whereas the last element points to the first bytecode to execute if this handler is chosen.

A remark on correctness. The above representation makes some implicit assumptions about the program. For example, the index of a method should be less or equal to the number of methods contained in the program. These assumptions are formalised as predicates on `jcprogram`. This is the (mild) price to pay for not using dependent types to represent programs.

2.3 The JCVM Tools

The transformation of JavaCard programs into cap files may be performed by standard tools, namely any Java compiler and JavaCard converter. In order to translate JavaCard programs into our format, we have developed a toolset, called the JCVM Tools (over 4,000 lines of Java code). The JCVM Tools transform a set of CAP files into a Coq expression of type `jcprogram`. In addition, the JCVM

Tools provide a graphical user interface to browse through programs and allow to modify compiled JavaCard programs (so as to check the behaviour of our formal semantics on incorrect programs). We have used the JCVM Tools to debug our formalisation.

3 The Virtual Machine

The Virtual Machine is described by a small-step semantics; more precisely, each instruction is formalised as a state transformer, i.e. a function that takes as input a state (before the instruction is executed) and returns a new state (after the instruction has been executed).

3.1 Values

In order to formalise the virtual machine, we first need to represent, for each JavaCard type, its possible values. These can either be arithmetic values or non-computational values such as memory addresses. Both can be represented as integers; for the latter, we use an implicit coercion from non-computational values to integers. As in [27], we tag values with their types. Formally, we set:[2]

```
Definition valu := type*Z.
```

Here `Z` is the (inductive) type of binary integers provided by the Coq library. While the inductive representation is suitable for reasoning (each integer has a unique representation in `Z`), it is less suited for computing and functions such as division are not part of the standard library. Besides, existing operations are not suitable to model overflows. In order to provide an accurate treatment of arithmetic, we therefore proceed as follows:

1. we introduce an alternative representation `Z_bits` of integers as lists of bits;
2. we define all arithmetic operations as functions over `Z_bits`. These functions abide to Sun's specifications for overflows;
3. we define bi-directional coercions between `Z` and `Z_bits` to switch between the two representations.

3.2 The Memory Model

States are formalised as triples consisting of the heap (containing the objects created during execution), the static heap (containing static fields of classes) and a stack of frames (environments for executing methods). Formally, states are defined by:

```
Definition jcvm_state := static_heap*heap*stack.
```

[2] The expression `value` is already used for the lift monad so we use `valu` instead.

The static heap is defined as a list of values, whereas the heap is defined as a list of objects. These can either be class instances or arrays, as formalized by the inductive (sum) type:

```
Inductive obj : Set :=
          Instance : type_instance -> obj |
          Array    : type_array -> obj.
```

Both `type_instance` and `type_array` are record types that contain all the relevant information for describing instances and arrays respectively. For example, a class instance is described by the index of the class from which the object is an instance, the instance variables (as a list of `valu`), the reference to the owning package and a flag to indicate whether the object is an entry point and whether it is a permanent or temporary entry point (entry points are used in the JavaCard security model for access control). Formally, we set:

```
Record type_instance : Set := {
  reference                : class_idx;
  contents_i               : (list valu);
  owner_i                  : Package;
  is_entry_point           : bool;
  is_permanent_entry_point : bool;
}.
```

Arrays are formalised in a similar fashion.

As to the stack, it is a list of frames that are created upon execution of a method and destroyed upon completion of the method's execution. Formally, we set:

```
Definition stack := (list frame).
```

Each frame has its own array of local variables and its own operand stack which is used to store a method's parameters and results. A frame also has a counter pointing to the next instruction to be executed, a reference to the current method, and a reference to the context of the current method (this context plays a fundamental role in the firewall mechanism). Formally, a frame is described by:

```
Record frame : Set := {
  locvars         : (list valu);      (* Local Variables *)
  opstack         : (list valu);      (* Operand stack *)
  p_count         : bytecode_idx      (* Program counter *)
  method_loc      : method_idx;       (* Location of the method *)
  context_ref     : Package;          (* Context Information *)
  analyzed_method : bool
}.
```

The `analyzed_method` is only used in Section 5 to define the abstraction function that maps each state to an abstract state; the abstraction function is itself used to express the correctness of the ByteCode Verifier.

3.3 Instructions

The semantics of each instruction is formalised using a function of type:

$$\texttt{jcvm\_state * operands} \rightarrow \texttt{returned\_state}$$

The type `operands` is not a Coq expression but a type determined by the instruction to be executed. In order to handle abrupt termination (that may arise because of uncaught exceptions), the codomain of the function is an inductive (sum) type:

```
Inductive returned_state: Set :=
  Normal   :       jcvm_state->returned_state |
  Abnormal : xcpt->jcvm_state->returned_state.
```

In case of normal execution, the returned state is the one obtained after execution of the instruction (tagged with `Normal`), whereas in the case of abrupt termination, the returned state is that of the virtual machine when the uncaught exception was raised (tagged with `Abnormal` and the nature of the uncaught exception). In order to execute the virtual machine, we collect the semantics of each instruction in a one-step execution function `exec_instr` of type:

$$\texttt{instruction*state*program} \rightarrow \texttt{returned\_state}$$

where `instruction` is the sum type of instructions. The function takes as inputs an instruction `i`, a state `s` and a program `p` and returns `sem_i s o` where `sem_i` is the semantics of `i` (of type `state*operands` $\rightarrow$ `returned_state`) and `o` is the list of operands required to execute the instruction (extracted from `state`).

Note that one cannot use `exec_instr` to build a function that takes as input a program and returns as output its result because Coq only supports terminating functions. However, we have used Coq's extraction mechanism successfully to derive a one-step execution function in CAML and wrapped it up with a while-loop to produce a certified JavaCard Virtual Machine.

3.4 Exception Management

JavaCard exceptions can either be raised by the program, via the instruction `athrow`, or by the virtual machine. In addition, execution may simply fail in case of an incoherence due to a memory problem, e.g. if a reference is not found in heap, or an execution problem, e.g. an empty stack for a `pop`. Our formalisation collects these three kinds of exceptions in an inductive (sum) type. Beware that exceptions in the virtual machine are represented as instances of exception classes, and not as inhabitants of the type `xpct`. In fact, we use the latter to give the semantics of exception handling.

We now turn to exception handling. Two situations may occur:

- the machine aborts. In the case of a `JCVMError`, the virtual machine is unable to continue the execution and, by calling an abort function, an abnormal state labelled by the error is returned;
- the exception handling mechanism is launched. In order to catch an exception, one searches for an adequate handler through the stack. This procedure is recursive (it is one of the few places where our formalisation uses recursion), see the full version of this paper.

3.5 Semantics of Invokevirtual

Most instructions have a similar execution pattern: (1) the initial state is decomposed; (2) fragments `F` of the state are retrieved; (3) observations `O` are made to determine the new state; (4) the final state is built on the basis of `O` and `F`. In this subsection we describe the semantics of `invokevirtual`. Because of space restrictions, we only consider the main function `new_frame_invokevirtual`. The function decomposes a state and creates a new frame for the method being invoked.

```
Definition new_frame_invokevirtual :=
[nargs:nat][m:Method][nhp:obj][state:jcvm_state][cap:jcprogram]
Cases state of
(sh, (hp, nil)) => (AbortCode state_error state) |
(sh, (hp, ((cons h lf) as s))) =>

  (* extraction of the list of arguments *)
  Cases (l_take nargs (opstack h)) (l_drop nargs (opstack h)) of
  (value l) (value l') =>

    (* security checking *)
    (if (violation_security_invokevirtual h nhp)
     then (ThrowException Security state cap)
     else

       (* then a signature check is performed *)
     (if (sig_ok l (signature m) cap)
       (* in case of success, the stack of frames is updated *)
     then (Normal (sh, (hp, (cons (Build_frame (nil valu)
                                               (make_locvars l (local m))
                                               (method_id m)
                                               (get_owner_context nhp)
                                               false
                                               (0))
                                  (cons (Build_frame l'
                                                     (locvars h)
                                                     (method_id_loc h)
                                                     (context_ref h)
                                                     (analyzed h)
                                                     (p_count h))
                                        (tail s))))))
        else (AbortCode signature_error state)
       )
    ) |
    _ _ => (AbortCode opstack_error state)
  end
end.
```

The function performs various security checks, including those imposed by JavaCard firewalls. E.g. the function `violation_security_invokevirtual` will verify, in case the object `nhp` is an instance, whether (1) the active context is

the JavaCard Runtime Environment context[3] or; (2) the active context is also the context of the instance's owner or; (3) the instance is an entry point. If not, the function returns `true` to flag a security violation.

4 Abstract Virtual Machine

When reasoning about the virtual machine and/or applications, it is convenient to omit some of its intricacies and consider simplified virtual machines instead. In this section, we develop such an abstract virtual machine that manipulates types instead of values. This abstract virtual machine represents, in some sense, a type-checking algorithm for the concrete virtual machine and indeed, in the next section, we show how to derive a ByteCode Verifier from this abstraction.

4.1 Abstract Memory Model

As a first approximation, we would like our abstract values to be the set of (JavaCard) types. However, return addresses needs a special treatment. In the semantics of the instruction `ret`, it is required that the first operand is a value `val` of type `Return_Address` and the integer part of `val` is used to indicate the next value of the program counter. If we simply forget about the integer part of `val`, we are simply unable to indicate the program point where execution is to be continued. We therefore adapt the definition of (abstract, JavaCard) types to store the value to which the program counter needs to be updated. Formally, abstract values (identified with) abstract types are defined as a mutual inductive type, together with abstract primitive types:

```
Inductive abs_type_prim : Set :=
                  abs_ReturnAddress : nat -> abs_type_prim | ...
```

The memory model is simplified likewise:

- the heap disappears. Indeed, the type of objects created during execution is always stored in the stack so the heap is not needed any longer.
- the stack disappears and is replaced by a frame. Indeed, execution may be performed on a method by method basis so that only the return type is required for executing a method's invocation (we return to the abstract semantics of invokevirtual in the next subsection). Hence we only need to consider one abstract frame instead of the stack.

We still need to maintain the static heap, abstracted as a list of types. The static heap is used for example in the semantics of `get_static`. Formally, we set:

```
Definition abs_jcvm_state := abs_static_heap*abs_frame.
```

We now turn to the execution model. Execution becomes non-deterministic because some branching instructions may return to different program points depending upon the value on top of the operand stack (and we do not have access to

[3] The JavaCard Runtime Environment is privileged and may access all objects.

the value). In order to handle this non-determinism, the corresponding abstract instructions are required to return a list of possible returned states. Formally, instructions are formalised as functions of type:

`abs_jcvm_state * operands` → `(list abs_returned_state)`

As for the concrete virtual machine, one defines a one-step abstract execution function `abs_exec_instruction` of type:

`instruction*abs_state*program` → `abs_returned_state`

4.2 Exception Management

The abstract virtual machine cannot handle standard JavaCard exceptions such as `NullPointer` or `Arithmetic` exceptions because they depend on values forgotten during the abstraction. In fact, the only exceptions handled by the abstract virtual machine are those caused by an incorrect program.

4.3 Semantics of the Abstract Invokevirtual

The abstract semantics for `new_frame_invokevirtual` does not create a new frame nor perform a check for security exceptions. Moreover the resulting state becomes:

```
(abs_Normal (sh, (Build_abs_frame
                      (app_return_type l' (Snd (signature m)))
                      (abs_locvars h)
                      (abs_method_loc h)
                      (abs_context_ref h)
                      (S (abs_p_count h)))))
```

The return type of the method called (if different from `Void`) is added to the operand stack of the current frame by calling the function `app_return_type` and the program counter is incremented.

4.4 Correctness

In order to state the correctness of the abstraction, we want to define a function that maps states to abstract states. As a first step, we define a function `alpha_val` mapping values to abstract values. Formally, we set:

```
Definition alpha_val [v:valu] : abs_valu :=
Cases (Fst v) of
 (Prim ReturnAddress) =>
                   (abs_Prim (abs_ReturnAddress (absolu (Snd v))))
|    _                => (type_to_abs_type (Fst v))
end.
```

where `absolu` coerces an integer to its absolute value and `type_to_abs_type` coerces a type to its corresponding abstract type.

Now we need to extend the function `alpha_val` to a function `alpha` that maps every state to an abstract state. It is a simple matter to extract an abstract static heap from a state, but some care is needed to extract an abstract frame from a stack. Indeed, we cannot map a stack to the abstraction of its top frame, because of the invokevirtual function (concrete execution creates a new frame whereas abstract execution does not). In order to cope with this situation, we use the flag of the current analysed frame `m`. If `m` is on the top of the stack then it is abstracted. If there are other frames above, the return type of the frame just above the analysed frame is added to the operand stack of `m` and `m` is then abstracted.

Finally, we extend the function `alpha` to a function `alpha_ret` that maps every returned state to an abstract returned state. The correctness of the abstraction is then stated as a "commuting diagram" relating concrete and abstract execution (up to subtyping), see Figure 1. The hooked vertical arrow on the right-hand side of the diagram and the $\leq$ sign at the bottom-right corner mean that the abstraction of the concrete returned state is, up to subtyping, a member of the list of abstract returned states.

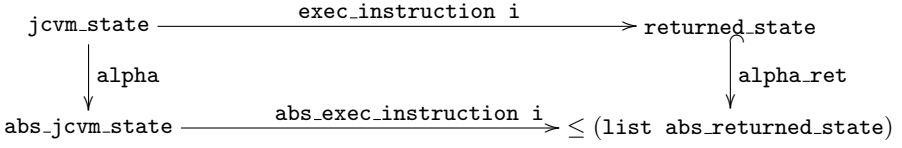

Fig. 1. Commutative diagram of concrete and abstract execution

We have shown[4] in Coq that the diagram commutes (up to subtyping), provided concrete execution does not raise any exception except by calling `AbortCode` (as discussed above, other exceptions cannot be handled by the abstract virtual machine). It follows that every call to `AbortCode` at the concrete level is matched at the abstract level. This is the key to proving the correctness of the ByteCode Verifier.

5 Application: A Certified ByteCode Verifier

The ByteCode Verifier is a key component of JavaCard's security. Below we present a certified ByteCode Verifier derived from the abstract virtual machine described in the previous section. Our ByteCode Verifier ensures that, at every

[4] Our proof assumes that the virtual machine verifies some basic properties w.r.t. memory management.

program point, the local variables and the operand stack are appropriately typed for the instruction to be executed. It also ensures that, if a program point is reached several times, the size and type of the operand stack must remain equal. Our ByteCode Verifier treats the whole set of instructions but is not complete: it does not treat subroutines in their full generality nor object initialisation.

5.1 The Verification Algorithm

All the properties suggested above, except the last one, can be verified by executing the abstract Virtual Machine. For the last property, we also have to record the (abstract) state of the (abstract) virtual machine after each step of execution. More precisely, we store an uninitialised returned state for each program point of the analysed method. After each step of execution, we check if the instruction has been performed before and if so, unify in a suitable way the returned state with the state that was stored for this instruction. In case of success, the resulting state after unification state is saved again. If, after unification, the saved state has not changed, the execution can stop for this particular execution path.

Some instructions require some extra care, e.g. (1) for instructions that can lead to two different program points, the execution must continue from both branching points; (2) for exception handlers, the `catch` block must be executed from the beginning with the appropriate arguments, and at the return point of the exception handler, an unification must occur.

5.2 Correctness of the ByteCode Verifier

The correctness of the ByteCode Verifier comprises two parts:

- a proof of termination. It requires to define a well-founded relation on types and to prove that each unification step produces a state that is strictly smaller than the state that was previously stored. The proof is highly non trivial and is used to define by well-founded recursion the ByteCode Verifier as a function `bcv:jcprogram` $\rightarrow$ `bool`;
- a proof of correctness. One needs to prove that, if bytecode verification is successful, then the function `AbortCode` will not be called. The proof, which uses the correctness of the abstraction, ensures that the ByteCode Verifier enforces the expected properties.

6 Conclusion

We have presented an executable formal semantics of the JavaCard Virtual Machine and ByteCode Verifier. With 15,000 lines of Coq scripts, our formalisation constitutes the most in-depth machine-checked account of the JavaCard platform to date.

6.1 Related Work

Applications of formal methods and programming language theory to Java and JavaCard are flourishing. Due to space constraints, we only comment on works that emphasise machine-checked verification of the Java or JavaCard platforms, either at platform or application level. Other works that do not emphasise machine-checked verification are discussed in the full version of this paper.

Platform-oriented projects. One of the most impressive achievements to date is that of the Bali project [2], which has formalised in Isabelle/HOL a large body of the Java platform, including (1) the type system and the operational semantics of both the source language and the bytecode, with a proof of type-safety at both levels; (2) the compiler, an abstract ByteCode Verifier and an abstract lightweight ByteCode Verifier, with a proof of their correctness; (3) a sound and complete axiomatic semantics to reason about Java programs. This work is comprehensive in that it treats all components of the Java platform, both at source and bytecode level, but does not take all aspects of Java (let alone JavaCard) into account. For example, Pusch's account [27] of the Java Virtual Machine does not handle arithmetic, exceptions, interfaces and initialisation and Nipkow's [24] and Klein and Nipkow's [19] accounts of the ByteCode Verifier focus on an even smaller fragment of the JVM. Thus, we see our work as complementary to theirs and as further evidence that, as suggested in [25], "machine-checking the design of a non-trivial programming language has become a reality".

Other partial formalisations of the Java and JavaCard platforms are reported by Y. Bertot [5] (object initialisation in Coq after [13]), by R. Cohen [10] (defensive JVM in ACL2), by T. Jensen and co-authors [29] (converter in Coq), by J.-L. Lanet and A. Requet [20] (JCVM in B), by Z. Qian and co-workers [9] (JVM and BCV in Specware) and by D. Syme [31] (operational semantics, type system and type soundness of source Java in DECLARE).

Application-oriented projects. Application-oriented projects may be further classified on the basis of the verification techniques used. These can either be mostly logical or mostly algorithmic.

Logical approaches. The LOOP tool [22], which allows to reason about (source) Java programs via a front-end to PVS and Isabelle, has been applied successfully to the verification of some standard Java class libraries and more recently to the JavaCard APIs. The key ingredients underlying the LOOP's approach are (1) a type-theoretical semantics of Java programs and of the Java memory model; (2) an axiomatic logic to reason about Java programs and Java features, including exceptions, abrupt termination and inheritance; (3) a tool to compile Java classes into PVS or Isabelle theories that form the basis for actual verifications. The main differences with our work are that their semantics works at source level and that it is not directly executable.

Rather similar techniques have been developed independently by A. Poetsch-Heffter and co-workers [23,26], while in [7], P. Brisset combines logical and algorithmic techniques to verify the correctness of Java's security manager. Further

uses of logical techniques for the verification of Java programs are reported in [1,11,14].

Algorithmic approaches. The Bandera project [3] has developed a toolset to verify automatically properties of (source) Java programs via a back-end to model-checkers such as SMV and Spin. The toolset has been successfully applied to verify properties of multi-threaded Java programs. The key ingredients underlying the Bandera's approach are (1) a (temporal logic like) specification language to describe program properties; (2) a toolset[5] that extracts from Java source code compact finite-state models; (3) an abstraction specification language and an abstraction engine that derives the abstract program for a given program and abstraction.

Further uses of algorithmic techniques to verify Java programs have been reported e.g. by P. Bieber and co-authors [6] (abstraction and model-checking to detect transitive flows in JavaCard), T. Jensen and co-workers [18] (abstraction and finite-state verification to verify control-flow security properties of Java programs), K. Havelund [15] (Java Path Finder, model-checking of concurrent Java programs), K. R. M. Leino and co-authors [21] (Extended Static Checking, with a back-end to the theorem-prover Simplify). In addition, numerous program analyses and type systems have been designed to verify properties of Java programs.

6.2 Future Work

Our primary objective is to complete our work into a full formalisation of the JavaCard platform (at the bytecode level) that may be used as a basis for reasoning about JavaCard programs. Clearly, much work remains to be done. Below we only outline the most immediate problems we intend to tackle.

Platform level. First and foremost, one needs to complete the formalisation of the ByteCode Verifier. The key challenge is of course to handle subroutines. We see two complementary options here: the first one is to provide a full account of subroutines along the lines of [13,30]. An alternative, first suggested by S. Freund in [12] and recently implemented in the KVM [8], would be to consider a program transformation that translates away subroutines and prove its correctness. Second, it would be interesting to extend our semantics with some features of Java, such as garbage collection, multi-threading and remote method invocation (RMI).

Application level. Many security properties can be expressed as temporal logic formulae over a program's execution trace and can in principle be verified by suitable algorithmic techniques. For these algorithmic verifications to be effective, they should be preceded by abstraction techniques that help reduce the state space. In this paper, we focused on the type abstraction which, in many

[5] The toolset combines several program analyses/program transformation techniques, including slicing and partial evaluation.

respects, underlies the ByteCode Verifier. We are currently trying to develop a method to generate automatically an abstract virtual machine and a proof of its correctness for any abstraction function mapping states to a suitably chosen notion of abstract states.

Acknowledgements. The authors would like to thank to Yves Bertot, Marieke Huisman, Jean-Louis Lanet, Shen-Wei Yu and the referees for their comments on this work. Simão Sousa is partially supported by a grant from the Portuguese Fundação para a Ciencia e a Technologia under grant SFRH/BD/790/2000.

References

1. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The Key approach: integrating design and formal verification of Java Card programs. In *Proceedings of the Java Card Workshop, co-located with the Java Card Forum, Cannes, France*, 2000.
2. BALI project. `http://www4.informatik.tu-muenchen.de/~isabelle/bali`
3. Bandera project. `http://www.cis.ksu.edu/santos/bandera`
4. B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, P. Loiseleur, C. Muñoz, C. Murthy, C. Parent-Vigouroux, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant User's Guide. Version 6.3.1*, December 1999.
5. Y. Bertot. Formalizing in Coq a type system for object initialization in the Java bytecode language. Manuscript, 2000.
6. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Electronic purse applet certification: extended abstract. In S. Schneider and P. Ryan, editors, *Proceedings of the workshop on secure architectures and information flow*, volume 32 of *Electronic Notes in Theoretical Computer Science*. Elsevier Publishing, 2000.
7. P. Brisset. A Case Study In Java Software Verification: SecurityManager.checkConnect(). In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, 5/2000, Fernuniversität Hagen, Fernuniversität Hagen, 2000.
8. CLDC and the K Virtual Machine (KVM). `http://java.sun.com/products/cldc`
9. A. Coglio, A. Goldberg, and Z. Qian. Towards a Provably-Correct Implementation of the JVM Bytecode Verifier. In *Formal Underpinnings of Java Workshop at OOPSLA*, 1998.
10. R. M. Cohen. Defensive Java Virtual Machine Specification Version 0.5. Manuscript, 1997.
11. M. Dam and P. Giambiagi. Confidentiality for mobile code: The case of a simple payment protocol. In *Proceedings of CSFW'00*. IEEE Press, 2000.
12. S. N. Freund. The Costs and Benefits of Java Bytecode Subroutines. In *Formal Underpinnings of Java Workshop at OOPSLA*, 1998.
13. 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.
14. R. Goré and L. Nguyen. CardKt: Automated Logical Deduction on Java Cards. In *Proceedings of the Java Card Workshop, co-located with the Java Card Forum, Cannes, France*, 2000.

15. 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.
16. JavaCard Forum. `http://www.javacardforum.org`
17. JavaCard Technology. `http://java.sun.com/products/javacard`
18. T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security policies. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 89–103. IEEE Computer Society Press, 1999.
19. G. Klein and T. Nipkow. Lightweight bytecode verification. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, 5/2000, Fernuniversität Hagen, Fernuniversität Hagen, 2000.
20. J.-L. Lanet and A. Requet. Formal Proof of Smart Card Applets Correctness. In *Proceedings of CARDIS'98*, 1998.
21. K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 251, 1999, Fernuniversität Hagen, Fernuniversität Hagen, 1999.
22. LOOP project. `http://www.cs.kun.nl/~bart/LOOP`
23. J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *Proceedings of TACAS'2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2000.
24. T. Nipkow. Verified Bytecode Verifiers. In F. Honsell, editor, *Proceedings of FOSSACS'01*, volume xxxx of *Lecture Notes in Computer Science*. Springer-Verlag, 2001. To appear.
25. T. Nipkow and D. von Oheimb. Java$_{light}$ is type-safe—definitely. In *Proceedings of POPL'98*, pages 161–170. ACM Press, 1998.
26. A. Poetzsch-Heffter and P. Müller. A Programming Logic for Sequential Java. In D. Swiestra, editor, *Proceedings of ESOP'99*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.
27. C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W. R. Cleaveland, editor, *Proceedings of TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 89–103. Springer-Verlag, 1999.
28. F. B. Schneider. Enforceable security policies. Technical Report TR99-1759, Cornell University, October 1999.
29. G. Ségouat. Preuve en Coq d'une mise en oeuvre de Java Card. Master's thesis, University of Rennes 1, 1999.
30. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proceedings of POPL'98*, pages 149–160. ACM Press, 1998.
31. 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.