

Gilles Barthe · Guillaume Dufay

# A Tool-Assisted Framework for Certified Bytecode Verification and its Application to Java Card

September 15, 2005

**Abstract** Bytecode verification is a key security function in several architectures for mobile and embedded code, and its formal correctness has been studied extensively, using dedicated or general purpose theorem provers.

The goal of our work is to develop methodologies, techniques and tools to help such formal endeavors. In a related effort, we have automated a methodology to establish the correctness of abstract virtual machines used by bytecode verification; however, this work did not consider how to construct a certified bytecode verifier from a certified abstract virtual machine. The purpose of this paper is to complete the process by certifying the different dataflow analyses involved in bytecode verification, using the Coq proof assistant. It enables us to derive automatically, and under minimal requirements, a provably correct bytecode verifier from a reference virtual machine that performs verification at run-time.

As an illustration of the benefits of our approach, we present an instantiation of our framework to derive the correctness of the Java Card platform, which is the standard *de facto* for new generation, multi-application smartcards.

---

## 1 Introduction

Virtual machines allow programs to abstract away from the specifics of the underlying hardware and operating system and have been brought to prominent use, notably in the context of mobile and embedded code. In order to prevent specific forms of programming errors to occur at run-time, virtual machines often have a safety policy, enforced statically through bytecode verification.

Bytecode verifiers (BCVs) are key security functions in several architectures, and in particular in the Java platform. Their role is to reject potentially insecure

programs that may violate type safety, or perform illegal memory accesses, or not respect the initialization protocol, or yield stack underflows or overflows, *etc.*, see e.g. [18]. Due to the importance of the bytecode verifier in the Java security architecture, and to the use of Java-enabled devices in security sensitive applications, the design and implementation of the BCV must be proved correct using proof assistants. Over the last few years, the correctness of Java bytecode verification, i.e. its adherence to the specification of Sun (or variants of it), has been proved formally by several projects, including industrial projects concerned with security evaluations for Java smartcards, see Subsection 4.2.

While such projects have been successful, providing machine-checked accounts of virtual machines and bytecode verifiers is labor-intensive, and could benefit from automated tool support for the most mundane tasks in developing machine-checked specifications and verifications of these languages. Our work aims at developing tools and libraries that help validate execution platforms for mobile and embedded code. The research reported here focuses on specifying and proving the correctness of bytecode verification, following a methodology that has been adopted by many projects, see e.g. [1, 3, 7, 17], and that factors the effort into two phases: a first, platform-specific phase concerned with virtual machines, and a second, generic phase, concerned with the construction of the bytecode verifier proper.

- *virtual machines phase (VM phase)*: during this first phase, one provides the specification of a defensive virtual machine that manipulates typed values and performs type-checking at run-time (as well as computations), and of an abstract virtual machine that only manipulates types and performs type-checking. Then one cross-validates these virtual machines, by showing that the abstract virtual machine detects all typing errors that may occur during the execution of a program on the defensive virtual machine. In previous work, we have been developing Jakarta, an environment which supports the specification and

cross-validation of the virtual machines [2], and offers a high level of automation for performing the VM phase. In a nutshell, Jakarta consists of a specification language JSL in which virtual machines can be described, an abstraction engine that transforms virtual machines (e.g. that extracts an abstract virtual machine from a defensive one), and an interface with the proof assistant Coq [10], which maps JSL specifications to the prover specification language and generates automatically correctness proofs for the cross-validation of virtual machines.

- *bytecode verification phase (BV phase)*: during this second phase, one builds and validates the bytecode verifier, using the abstract virtual machine defined during the VM phase. It involves modeling a dataflow analysis for an unspecified execution function that meets minimal requirements, and proving its correctness; essentially it amounts to showing that the analysis will reject all programs that go wrong during execution. Bytecode verifiers come in several flavors, according to the policy to be verified, the degree of precision required, and the resources that are available for bytecode verification; for example, so-called polyvariant analyses are used so to account for some specificities of the Java platform such as subroutines. In order to cover all possible bytecode verifiers, the BV phase must therefore show the correctness of the family of dataflow algorithms that are used for bytecode verification.

The purpose of this paper is to present a modular framework for performing the BV phase. Starting from an abstract notion of virtual machine on which we only impose minimal assumptions, we build a parametric bytecode verifier that encompasses a large class of algorithms for bytecode verification, and show the verifier to be correct in the sense that it will reject programs that may go wrong. One novelty of our framework is to provide a high-level proof that it is sound to perform bytecode verification on a method per method basis. Another novelty is to provide a generic bytecode verifier that can be instantiated to several analyses including standard analyses and analyses for which no formal correctness proof was previously known.

The combination of our framework for bytecode verification with the Jakarta toolset yields an automated procedure to derive a certified bytecode verifier from a reference defensive virtual machine. Our procedure is applicable to many settings; as a realistic example of its application, we show how to instantiate our generic framework to certify the Java Card platform, a dialect of Java adapted to the resource constraints and security requirements of smartcards, and the standard programming language for smartcard applications.

*Contents of the paper* The remaining of the paper is organized as follows. In Section 2, we explain the basis for bytecode verification and describe our bytecode verifier

framework. In Section 3, we instantiate the framework to the Java Card virtual machine. We conclude in Section 4 with related work, a general perspective on our results thus far, and directions for further research. As an appendix, we give some reminders of the Coq syntax used throughout the paper.

---

## 2 Bytecode Verification

### 2.1 Principles and algorithms

Bytecode verification [11, 18] is a static analysis that is performed method per method on compiled programs prior to their loading. Its aim is to reject programs that violate type safety, perform illegal memory accesses, do not respect the initialization protocol, yield stack underflows or overflows, *etc.*

The most common implementation of bytecode verification is through a dataflow analysis [14] instantiated to the abstract virtual machine that operates at the type level. The underlying algorithm relies on a history structure, storing the computed abstract states for each program point, and on a unification function on states. Then, starting from the initial state for a method, it computes a fixpoint with the abstract execution function. If the error state does not belong to the resulting history structure then bytecode verification is successful.

In the standard algorithm, called monovariant analysis, the history structure only stores one state for each program point and the unification function unifies, performing a join on the Java Card type lattice, the computed state (resulting from one step of abstract execution) and the stored state. Unfortunately, this algorithm does not accept polymorphic subroutines (subroutines called from different program points). To handle such subroutines, the history structure must contain a set of states for each program point. For the polyvariant analysis, the unification function adds the computed state to the corresponding set from the history structure. This technique needs much more memory than monovariant analysis, however, it is possible to perform state unification rather than set addition in most cases. This last technique, called hybrid analysis (as described in [9, 13]), offers the best compromise between memory consumption, precision and efficiency.

Our framework also deals with lightweight bytecode verification [21], a special kind of verification that can fit and run in chips used for smart cards, but due to space constraints details are omitted.

### 2.2 Bytecode Verification as a Fixpoint Computation

*Notations* In order to simplify the definitions used throughout this section, we introduce some notations. The types predicate  $A$  and relation  $A$  respectively denote the

set of predicates and binary relations over a type  $A$ . Given  $A : \text{Set}$ ,  $<_A : (\text{relation } A)$ ,  $f : A \rightarrow A$  and  $P : (\text{predicate } A)$ , we let  $\leq_A$  denote the reflexive closure of  $<_A$  and define:

$$\begin{aligned} (\text{monotone } <_A f) &\equiv \\ \forall (a, a' : A), a <_A a' &\rightarrow (f a) \leq_A (f a') \\ (\text{decreases } <_A f) &\equiv \\ \forall (a : A), (f a) &\leq_A a \\ (\text{down\_closed } <_A P) &\equiv \\ \forall (a, a' : A), a <_A a' &\rightarrow (P a') \rightarrow (P a) \end{aligned}$$

Finally, we let  $(\text{well\_founded } <_A)$  state that the relation  $<_A$  is well founded, i.e. that there is no infinite decreasing chain.

*Fixpoint Computation* We favor a definition of a bytecode verifier that abstracts away from implementation details, and define it as a predicate that rejects programs that may go wrong. The latter being defined from a transition system.

#### Definition 1

A *transition system with error* (TSE) is given by a type state of *states*, an *execution relation*  $\text{exec}$  over states, and a set  $\text{err}$  of error states. Formally:

```
Module Type TSE.
Parameter state : Set.
Parameter exec  : (relation state).
Parameter err   : (predicate state).
End TSE.
```

We say that a state  $a$  of a given TSE is bad, written  $\text{bad } a$ , if it can reach an error state by successive transitions of the execution relation.

**Definition 2** A *bytecode verifier* over a module  $\text{tse}$  of type TSE is given by a predicate  $\text{check}$  that rejects all bad states. Formally, the module  $\text{BCV}$  of bytecode verifiers extends the module TSE as follows<sup>1</sup>:

```
Module Type BCV.
Declare Module tse : TSE. Import tse.
Parameter check : (predicate state).
Axiom  $\forall a : \text{state}, (\text{check } a) \rightarrow \neg(\text{bad } a)$ .
End BCV.
```

The standard way to build a bytecode verifier is to endorse the type of states with a well-founded order for which execution is decreasing (to guarantee termination), and such that error states are downwards closed. If furthermore execution is deterministic, one can compute for every state  $a$ , the greatest fixpoint  $b$  below  $a$ ; then it is sufficient to check that  $b$  is not an error state to conclude that  $a$  is not bad.

**Definition 3** A *fixpoint structure with errors* (FSE) is given by the module

```
Module Type FSE.
Parameter state : Set.
Parameter exec  : state  $\rightarrow$  state.
Parameter err   : (predicate state).
Parameter <_state : (relation state).
```

```
Axiom (well_founded <_state).
Axiom (decreases <_state exec).
Axiom (monotone <_state exec).
Axiom (down_closed <_state err).
End FSE.
```

We can define a module functor satisfying, from a module of type FSE, the type of the module BCV:

```
Module FSE2BCV [fse : FSE] <: BCV.
```

To do so, we first define for every state  $a$  of a FSE the greatest fixpoint  $\text{gfp } a$  below it as:

$$\text{gfp } a = \begin{cases} a & \text{if } \text{exec } a = a \\ \text{gfp } (\text{exec } a) & \text{otherwise} \end{cases}$$

Then, we define  $\text{check } a$  as  $\neg(\text{err\_state } (\text{gfp } a))$ . As execution is monotone and  $\text{gfp } a$  is the greatest fixpoint below  $a$ , it is clear that such a checking is sufficient to guarantee that  $a$  is not a bad state.

### 2.3 A Parameterized Bytecode Verifier

In this subsection, we construct a parameterized bytecode verifier that rejects programs that may go wrong when executed with an abstract virtual machine. We start with the definition of the latter.

**Definition 4** An abstract virtual machine (AVM) is given by an ordered type of states  $\text{state}$  endorsed with a downwards closed set of errors  $\text{err}$ , a type of locations  $\text{loc}$ , an execution function  $\text{exec}$ , a successor function  $\text{succs}$  that computes the successors of a state and an enumeration  $\text{locs}$  of the locations of the program. Formally:

```
Module Type AVM.
Parameter state : Set.
Parameter <_state : (relation state).
Parameter err   : (predicate state).
Parameter loc   : Set.
Parameter succs : loc  $\rightarrow$  state  $\rightarrow$  (list loc).
Parameter locs  : (list loc).
Parameter exec  : loc  $\rightarrow$  state  $\rightarrow$  state.
```

```
Axiom (down_closed <_state err).
End AVM.
```

Bytecode verification relies on stackmaps, i.e. functions that associate to every program point a history structure. History structures can be seen as an abstraction of the mathematical set notion.

**Definition 5** The module type  $\text{History\_Struct}$  of *history structures* is parameterized by a carrier set  $A^2$  and

<sup>1</sup> Coq modules provide names for axioms, so that these axioms can later be used in proofs. For readability we omit names of axioms in our module declarations.

<sup>2</sup> In reality, Coq modules type can only be parameterized by other modules, so one has to use a module that is “isomorphic” to  $\text{Set}$ .

given a type constructor `hist`, a projector function `hist_unit`, an extension `hist_less` of an order on  $A$ , an decreasing iterator `hist_foldr` and a membership predicate  $\in_{\text{hist}}$  on which we define an existential predicate  $\exists_{\text{hist}}$ . Formally:

```
Module Type History_Struct [A:Set].
Parameter hist      : Set → Set.
Parameter hist_unit : A → (hist A).
Parameter hist_less :
  ∀ <_A : (relation A), (relation (hist A)).
Parameter ∈_hist   : A → (hist A) → Prop.
Axiom ∀ (x:A), (x ∈_hist (hist_unit x)).
```

```
Parameter hist_foldr : ∀ (B : Set),
  (A → B → B) → B → (hist A) → B.
Axiom ∀ (B : Set) (f : (A → B → B))
  (<_B : (relation B)),
  (∀ a : A, (decreases <_B (f a))) →
  (∀ a : (hist A),
    (decreases <_B (fun (b : B) ⇒ (hist_foldr B f b a)))).
```

```
Definition ∃_hist :=
  fun (P : (predicate A)) (s : (hist A)) ⇒
    ∃ a, (P a) ∧ (a ∈_hist s).
Axiom ∀ <_A : (relation A) (P : (predicate A)),
  (down_closed <_A P) →
  (down_closed (hist_less <_A) (∃_hist P)).
End History_Struct.
```

In the following, we will use the notation  $\lt_A^{\text{hist}}$  for the extension (`hist_less <_A`) of a relation  $\lt_A$ .

The definition of stackmaps is included in a module of stackmap structures. Essentially, a stackmap structure consists of an abstract virtual machine, of a history structure, and of a unification function that merges states and that is decreasing and monotone w.r.t. the order inherited from the history structure. In order to construct a fixpoint structure (FSE) from a stackmap structure, we also require histories  $\lt_A^{\text{hist}}$  to be well-founded and a supremum for  $\lt_A$ .

**Definition 6** The module `Stackmap_Struct` of *stackmap structures* is defined as:

```
Module Type Stackmap_Struct.
Declare Module avm : AVM.
Declare Module hs : (History_Struct state).
Import avm. Import hs.

Parameter unify :
  state → (hist state) → (hist state).
Axiom ∀ (s:state), (decreases <_state^hist (unify s)).
Axiom ∀ (s:state), (monotone <_state^hist (unify s)).
Axiom ∀ (s:state) (s':(hist state)),
  (unify s s')=s' →
  ∃ y, (y ≤_state s) ∧ (y ∈_hist s').

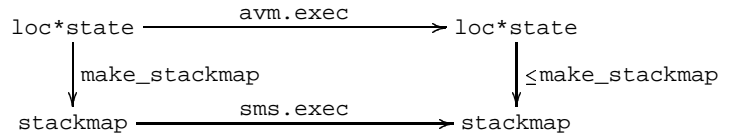
Parameter T : state.
Axiom ∀ (a:state), (a ≤_state T).

Axiom (well_founded <_state^hist).
End Stackmap_Struct.
```

One can define the type `stackmap` of stackmaps over a stackmap structure `sms` as `list (sms.avm.loc *(sms.hs.hist sms.avm.state))`, and an execution function

over stackmaps `sms_exec : stackmap → stackmap` which corresponds to the recursive procedure in Kildall's algorithm [14]. It is straightforward to derive a fixpoint structure, and hence a bytecode verifier `BCV`, for the TSE induced by `sms.exec`.

In order to conclude that the resulting fixpoint structure also yields a bytecode verifier for the TSE induced by the AVM, one needs to observe that the following diagram commutes:



Here  $\leq$  is the order on stackmaps and `make_stackmap` denotes the function that takes as input a pair  $(l, a)$ , and returns as output the stackmap in which the program point  $l$  is associated to the singleton history `hist_unit a`, and every other program point is associated to  $\top$ .

## 2.4 Instantiation Of History Structures

In the previous subsection, we have shown that a module of type `Stackmap_Struct` is sufficient to build a bytecode verifier for the AVM. The purpose of this subsection is to present different instantiations of our framework, focusing on different choices of history structures that correspond to the algorithms described in Subsection 2.1. We present new module types and corresponding modules functors to a `Stackmap_Struct`, that can be used as convenient entry points in our formalization.

### 2.4.1 Monovariant Analysis

A *monovariant analysis* is given by a well-founded order on states with a supremum, and by proofs that the execution function is monotone w.r.t. the order on states and the unification is decreasing and monotone. Formally:

```
Module Type Monovariant_Analysis_Struct.
Declare Module avm : AVM. Import avm.

Parameter unify : state → state → state.

Axiom ∀ (s:state), (decreases <_state (unify s)).
Axiom ∀ (s:state), (monotone <_state (unify s)).
Axiom ∀ (s,s':state), (unify s s')=s' →
  ∃ y, (y ≤_state s) ∧ (y ∈_hist s').

Parameter T : state.
Axiom ∀ (a:state), (a ≤_state T).

Axiom (well_founded <_state).
Axiom ∀ (l:loc), (monotone <_state (exec l)).
End Monovariant_Analysis_Struct.
```

The construction of a stackmap structure from a monovariant analysis is done by a functor module from the previous module type definition. It mainly proceeds by instantiating the parametric history structure to the identity history structure, in which  $\text{hist } A$  is defined as  $A$ , and the other fields are instantiated in the obvious way.

### 2.4.2 Polyvariant Analysis

A *polyvariant analysis* is given by a natural number  $\text{max\_length\_set}$  that fixes the maximal size of the set of abstract states associated to each program point, by a supremum state  $T$ , by an error state  $\text{err\_st}$  and by a proof that execution is monotone. Formally:

```
Module Type Polyvariant_Analysis_Struct.
Declare Module avm : AVM. Import avm.
```

```
Parameter max_length_set : nat.
Parameter err_st : state.
Axiom (err err_st).
Parameter T : state.
Axiom  $\forall (a:\text{state}), (a \leq_{\text{state}} T)$ .
```

```
Axiom  $\forall (l:\text{loc}), (\text{monotone } <_{\text{state}} (\text{exec } l))$ .
End Polyvariant_Analysis_Struct.
```

One proceeds by instantiating the history structure in such a way that  $\text{hist } A$  is defined as the set of elements of  $A$  of cardinal less than  $\text{max\_length\_set}$  (the other fields are instantiated in the obvious way). Then, this module is used with the functor `Polyvariant_Analysis` to construct a stackmap structure, defining the function `unify` as:

```
fun (a : state) (s : (hist state))  $\Rightarrow$ 
(if ((set_size (set_add a s)) < max_length_set
)
then (set_add a s)
else (set_add err_st s))
```

In that case, `hist_less` does not use the order  $<_{\text{state}}$  and is defined as set inclusion. It is interesting to notice that the polyvariant analysis is by far the simplest algorithm to instantiate.

### 2.4.3 Hybrid Analysis

An *hybrid analysis* is given combining elements needed by monovariant and hybrid analysis and adding an optimization function `opt_unify` to discriminate in which cases the unification of states must take place. Formally:

```
Module Type Hybrid_Analysis_Struct.
Declare Module avm : AVM. Import avm.
```

```
Parameter opt_unify : state  $\rightarrow$  state  $\rightarrow$  bool.
Parameter unify : state  $\rightarrow$  state  $\rightarrow$  state.
```

```
Axiom  $\forall (s:\text{state}), (\text{decreases } <_{\text{state}} (\text{unify } s))$ .
Axiom  $\forall (s:\text{state}), (\text{monotone } <_{\text{state}} (\text{unify } s))$ .
Axiom  $\forall (s,s':\text{state}), (\text{unify } s s')=s' \rightarrow$ 
 $\exists y, (y \leq_{\text{state}} s) \wedge (y \in_{\text{hist}} s')$ .
```

```
Parameter max_length_set : nat.
Parameter err_st : state.
Axiom (err err_st).
Parameter T : state.
Axiom  $\forall (a:\text{state}), (a \leq_{\text{state}} T)$ .
```

```
Axiom (well_founded  $<_{\text{state}}$ ).
Axiom  $\forall (l:\text{loc}), (\text{monotone } <_{\text{state}} (\text{exec } l))$ .
End Hybrid_Analysis_Struct.
```

The same history structure as polyvariant analysis is used for the hybrid analysis. The function `unify` is then defined as follows:

```
fun (a : state) (s : (hist state))  $\Rightarrow$ 
match (set_map_bool opt_unify unify a s) with
| (Some res)  $\Rightarrow$  res
| None  $\Rightarrow$ 
  (if ((set_size (set_add a s)) <
max_length_set)
then (set_add a s)
else (set_add err_st s))
end
```

where `set_map_bool` ranges over elements of  $s$ , performs unification depending on the result of `opt_unify` and returns the resulting set if unification has occurred or `None` otherwise. In that case, `hist_less` combines the order  $<_{\text{state}}$  on states and set inclusion.

## 2.5 Correctness Of Bytecode Verification

In the Subsection 2.3, we have shown that programs that pass bytecode verification do not go wrong when executed on an abstract virtual machine which satisfies minimal requirements. The purpose of this subsection is to lift this result to a defensive virtual machine. More precisely, we are going to show that programs that pass bytecode verification do not go wrong when executed on a defensive virtual machine which satisfies minimal requirements. It involves relating a defensive and an abstract virtual machine, and proving that no difficulty arises through exception handling (which is performed by the defensive virtual machine, but ignored by the abstract one), or through method invocation (which remains within the same frame for the abstract virtual machine, as explained below). We begin by defining defensive virtual machines.

**Definition 7** A *defensive virtual machine* (DVM) is given by a type `state` of states, an execution function `exec`, a type `frame` of frames, an accessor function `getstack` that associates to each state a list of frames (i.e. its stack), another accessor function `getinstr` that associates to each state the nature of the next execution to be executed, and a set `err_frame` of error frames. Formally:

```
Module Type DVM.
Parameter state : Set.
Parameter exec : state  $\rightarrow$  state.
Parameter frame : Set.
Parameter getstack : state  $\rightarrow$  (list frame).
```

**Parameter** `getinstr` : `state`  $\rightarrow$  `type_of_instr`.  
**Parameter** `err_frame` : (`predicate frame`).  
**End** `DVM`.

The function `getinstr` distinguishes between 4 cases: execution is intra-procedural `sameframe` (that acts only in the current frame, e.g. for arithmetic instruction, branching instruction, *etc*), execution is a method invocation `invoke`; execution is a return step `return` (pops a frame); or execution raises an exception `exception`.

In the following, we will formulate a set of general properties about method invocation and exception handling, and prove that such properties ensure that programs that pass bytecode verification will not go wrong. These properties involve an abstract virtual machine and an abstraction function.

*Abstract Virtual Machine* We assume given an abstract virtual machine `avm`, with a function `init` that returns `s` for each method or exception the corresponding initial state. Furthermore, we assume given a decomposition of abstract method invocation into two functions, so as to be able to simulate the modifications made by the concrete virtual machine on a frame when the control flow is given to the invoked method and when it returns to the invoker method. Formally, these two functions `exec_invk` and `exec_ret` are such that their composition is equal to `avm.exec` for states `a` such that `getinstr a = invoke`.

*Abstraction Function* We assume given a function that maps a frame to an abstract state and a location  $\alpha : \text{dvm.frame} \rightarrow \text{avm.state}$ . The function is extended a function  $\beta : \text{dvm.state} \rightarrow \text{avm.state}$  on defensive states by abstracting the topmost frame of the stack (if the stack is empty, we return a default error value).

*Safe States* We now turn to the definition of safe abstract states. A abstract state will be safe if it is greater than a state belonging to the history structure computed by the abstract bytecode verifier at the location of the given state. This notion is extended to defensive frames by abstraction.

The notion of safety for a defensive state must guarantee that the stack is well-formed, i.e. that all the frames below the top one are in an “intermediate” state which is not reached by the abstract virtual machine until the invoked method returns. Then, a defensive state `s` will be safe if `getstack s = []` or if `getstack a = f::lf` and each frame in `lf` is of the form `exec_invk f'` where `f'` is a safe frame.

We now show that safe states are closed under execution and are not bad, using cross-validation results between defensive and abstract virtual machines.

**Lemma 1** *Let `s` be a defensive state. Suppose:*

- *if `getinstr s = sameframe`, then the following property holds:*

$$(\text{avm.exec } (\beta \ s)) \leq_{\text{state}} (\beta \ (\text{dvm.exec } s))$$

- *if `getinstr s = invoke` and `getstack s = f::lf`, then the following properties hold:*

$$\begin{aligned} \exists f' : \text{frame}. \text{getstack } (\text{dvm.exec } s) = f' : : (\text{exec\_add } f) : : \text{lf} \\ (\text{init } (\beta \ (\text{dvm.exec } s))) \leq_{\text{state}} (\beta \ (\text{dvm.exec } s)) \end{aligned}$$

- *if `getinstr s = return` and `getstack (dvm.exec s) = f::lf`, then there exists two frames `f'` and `f''` such that the following properties hold:*

$$\begin{aligned} \text{getstack } s = f' : : f'' : : \text{lf} \\ (\text{aexec\_ret } (\alpha \ f'')) \leq_{\text{state}} (\alpha \ f) \end{aligned}$$

- *if `getinstr s = exception` and `getstack s = f::lf`, then the following properties hold:*

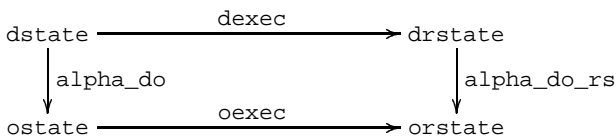
$$\begin{aligned} \exists f' : \text{frame}. \text{getstack } (\text{dvm.exec } s) = f' : : \text{lf}' \\ (\text{init } (\beta \ (\text{dvm.exec } s))) \leq_{\text{state}} (\beta \ (\text{dvm.exec } s)) \end{aligned}$$

*If furthermore `s` is safe, then `dvm.exec s` is also safe.*

This lemma if proved using properties on the bytecode verifier and property on `exec_invk` and `exec_ret` w.r.t. `avm.exec`. Then one can easily construct a bytecode verifier for a defensive virtual machine `dvm`. Formally, from a module type `Comp_Struct`, that contains all the assumptions of Lemma 1, we are able to define a functor module `BCV_dexec` satisfying the module type `BCV` for the execution `dvm.exec`. The function `check` of the module type `BCV` is defined assuming that the given defensive state is safe and that the result of bytecode verification for all initial states (methods and exceptions) of the program does not contain an error state. Finally, by Lemma 1, we can prove the property `check_ok` of the module type `BCV`, stating that if the verification check was successful, we can not reach with the defensive virtual machine an error state.

## 2.6 An Offensive Virtual Machine

As a corollary of the correctness of bytecode verification w.r.t the defensive virtual machine, it is possible to build a virtual machine that does not perform checks required from the defensive virtual machine but which is as safe as the latter provided bytecode verification has been successful. Such a virtual machine, faster for execution and closer to a real implementation, is called offensive. We only require from such a virtual machine that its execution function matches the defensive virtual machine one’s when the latter does not lead to an error. This cross-validation result can be expressed by the commutation of the diagram of Figure 1.



**Fig. 1** COMMUTATIVE DIAGRAM OF DEFENSIVE AND OFFENSIVE EXECUTION

As a main result, we prove that offensive and defensive virtual machines coincide on programs that pass bytecode verification. Formally:

$$\forall (s:dvm.state), (check\ s) \rightarrow (ovm.alpha\ (dvm.exec\ s)) = (ovm.exec\ (ovm.alpha\ s)).$$

where `check` is provided by the module `BCV_dexec`, `ovm` represents the offensive virtual machine (the corresponding module type is similar to `DVM`) and `ovm.alpha` is the abstraction function between defensive and offensive states.

### 3 Instantiation to the Java Card platform

Java Card is a popular programming language for open platform smart cards. According to the Java Card Forum, Java Card is the ideal choice for smart cards because Java Card developers can benefit from the well-established Java technology.

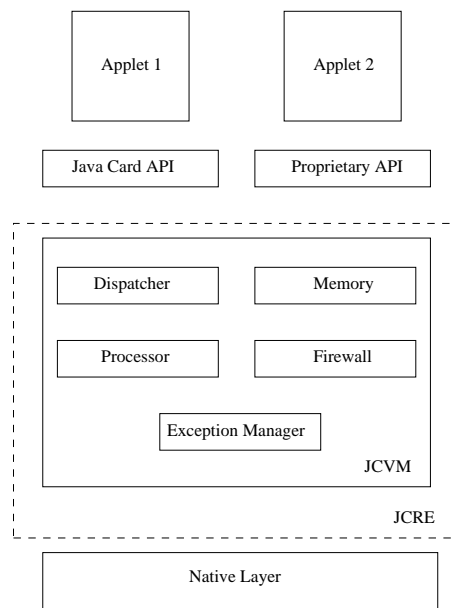
In fact, Java Card programs are written in a Java, however, due to limited memory resources and computing power, not all the language features defined in the Java Language Specification are supported on the Java Card. Specifically, the Java Card does not support dynamic class loading, threads, cloning, finalization, large primitive data types.

Java Card program may use APIs such as Java Card APIs, industry-specific APIs (e.g. GSM APIs) and proprietary APIs (e.g. Global Platform). The Java Card APIs provide support for runtime features that are not present or differently implemented in Java, in particular

- persistent and transient objects;
- atomic transactions;
- applet firewall and object sharing.

Applets must be verified (such as described in Section 2) and loaded on the card. On-card, the Java Card Runtime Environment (JCRE) manages applet installation and applet execution. The JCRE, whose architecture is shown in Figure 2, includes the Java Card Virtual Machine JCVM, the implementation of the Java Card APIs, and invokes the services of a native layer to perform such low-level tasks as communication management.

In the following, we will describe how to instantiate the framework presented in Section 2 to the Java Card

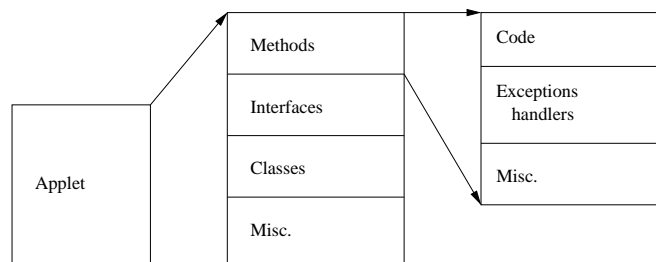


**Fig. 2** THE JCRE ARCHITECTURE.

platform. To this end, we need to provide modules of the types described in Section 2 for the Java Card platform. For instance, we need a module `DJCVM` of type `DVM` for the defensive Java Card virtual machine. First, we will give an overview of the representation of Java Card programs, that constitute a parameter of the execution functions of the virtual machines.

#### 3.1 Representation Of Java Card Programs

In this subsection, we present the Coq formalization of Java Card programs. Our virtual machines does not work directly on the binary representation of the CAP file format but on structured representation of programs. The latter is obtained from the former by a dedicated tool. After this transformation, only the essential components of an typed object-oriented language remain: types, methods, classes and interfaces. Figure 3 gives an overview of our representation of programs.



**Fig. 3** FORMALIZATION OF APPLETS.

### 3.1.1 Type system

The Java Card Virtual Machine distinguishes between primitive types and reference types. Primitive types are either arithmetic types (Byte, Short, Int) or Boolean or ReturnAddress types. ReturnAddress stands for the type of the address of an instruction and is used for going back from a subroutine (in the jsr and ret instructions that respectively jump into subroutine and returns from subroutine). There is also a special non-dynamic type named void, only used within signatures of methods. Formally, the Coq type that introduces Java Card primitive types is defined by:

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

There are four kinds of reference types that are null types, array types, instance types and interface types. Arrays are described according to the type of their elements which must be primitive or reference. The reference of an instance or an interface corresponds to a natural number which is the position in the lists of the classes or interfaces of the program (Section 3.1.2). Finally, the type of Java Card types is given by the following mutual inductive definition that captures the considerations below:

```
Inductive type : Set :=
| Prim : type_prim → type
| Ref : type_ref → type
with type_ref : Set :=
| Ref_null : type_ref
| Ref_array : type → type_ref
| Ref_instance : class_idx → type_ref
| Ref_interface : interf_idx → type_ref.
```

### 3.1.2 Programs

As already said, our program model does not contain any constant pool, contrarily to the binary representation of CAP files. We will suppose the linking phase already performed. Thus, the definition of a Java Card program in our formalization is limited to a collection of classes, interfaces and methods, and the description of the types of static values (for initialization purposes):

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

A method is characterized by its status (static or not), its signature (against which one can type-check its arguments upon invocation), the number of its local variables (for initializing its execution context), its list of

instructions to be executed (the bytecode), its exception handlers (a handler is an object that identifies the code for managing dynamic errors), its maximum operand stack size and finally its owning class and the indexes of the method. Formally, we use the following structure to represent methods:

```
Record Method : Set := {
  is_static : bool;
  is_init : bool;
  signature : ((list type)*type);
  nb_local : nat;
  bytecode : (list Instruction);
  handler_list : (list handler_type);
  m_max_opstack_size : nat;
  owner : class_idx;
  method_idx : method_idx
}.
```

where class\_idx and method\_idx (defined as natural numbers) corresponds respectively to the type of indexes to a class and to a method.

Classes (Class) and interfaces (Interface) are represented in a similar way (with records).

Finally, the type Instruction enumerates with an inductive type the different bytecodes of the Java Card language, with their operands:

```
Inductive Instruction : Set :=
| nop : Instruction
| push : type_prim → Z → Instruction
| ret : locvars_idx → Instruction
| invokespecial : nat → method_idx → Instruction
| getfield : type → field_idx → Instruction
| putstatic : type → field_idx → Instruction
| inc : type_prim → Z → nat → Instruction
...
```

Some bytecodes with similar semantics have been collapsed into a single one. For instance, the bytecode inc takes as a supplementary argument a primitive type and thus represents the bytecodes iinc et sinc of Java Card (incrementation of a register of type Int and Short respectively). Thus, it is possible to represent the 185 Java Card bytecodes by only 44 bytecodes in our formalization.

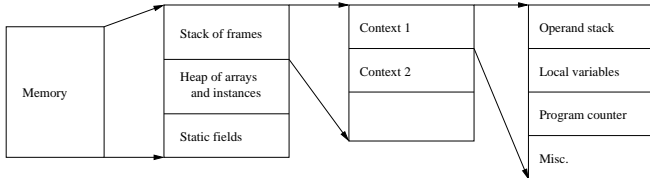
## 3.2 The Defensive Virtual Machine

The virtual machine described hereafter is a defensive virtual machine. It follows the specification of the reference virtual machine described by Sun in [24,25] and explicitly implements all the pre-conditions (the *must* and *must not* clauses) stated by the official Sun specification.

Each instruction of the virtual machine is given by a small-step semantics; more precisely, each instruction is formalized 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).

### 3.2.1 Runtime Environment

A state of a program contains all the memory items manipulated by this program: values, heap, stack of frames. Figure 4 gives an overview of our representation of the runtime environment.



**Fig. 4** FORMALIZATION OF THE RUNTIME ENVIRONMENT.

*Values* Values constitute the main element manipulated by the virtual machine. Values are typed and follow a definition similar to the type system, with an inductive type, where we distinguish primitive and reference values. Type information is carried by the constructor and the numerical value by the argument of the constructor. We obtain:

```
Inductive val_prim : Set :=
| vReturnAddress : bytecode_idx → val_prim
| vBoolean : Z → val_prim
| vByte : Z → val_prim
| vShort : Z → val_prim
| vInt : Z → val_prim.
```

A reference value is either null, an array or an instance, it is then defined as follows:

```
Inductive val_ref : Set :=
| vRef_null : val_ref
| vRef_array : type → heap_idx → val_ref
| vRef_instance : class_idx → heap_idx → val_ref.
```

where `heap_idx` indicates the location of an object in memory. There is no value corresponding to interfaces, since an interface must be implemented by a class to be used, through an instance.

Finally, the Coq type of Java Card values is defined by:

```
Inductive val : Set :=
| vPrim : val_prim → val
| vRef : val_ref → val
| vNonInit : class_idx → bytecode_idx → heap_idx → val.
```

where `vNonInit` is a specific value of non-initialized class instances.

*Objects* Objects in memory constitute the runtime representation of classes and arrays. The `vRef_instance` and `vRef_array` values defined in the previous subsection refer to these objects. We collect in a record information needed for each of these types.

For a class instance, we precise the instantiated class, the content of class variables, the owning context (for the security mechanisms) and a flag that precise whether the instance has been declared as an entry point (accessible from any context):

```
Record type_instance : Set := {
  reference : class_idx;
  contents_i : (list val);
  owner_i : AID;
  ptE : status_ptE
}.
```

Arrays are formalized in a similar fashion. Then, we define an inductive type for objects:

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

Finally, the memory area in which objects are stored during the execution of a program, the heap, is naturally defined as:

```
Definition heap := (list obj).
```

*Frames and Stack* When a method is called, an execution context for the method called, a frame, is created and added to the top of existing frames stack. Frames contains computational information: an operand stack and a set of local variables (or registers). An index records the referring method, and a program counter points to the next instruction to execute within the method. Finally, the maximum operand stack size is duplicated within the frame to avoid lookup in the method body.

```
Record frame : Set := {
  opstack : (list val);
  locvars : (list (option val));
  method_loc : method_idx;
  p_count : bytecode_idx;
  context_ref : AID;
  max_opstack_size : nat
}.
```

The type option of the field `locvars` will be used to formalize non initialized local variables.

In the definition of the module `DJCVm`, the parameter `frame` from the module `type DVM` is naturally instantiated to the previous declaration.

The frames stack (stack for short) is then simply defined with a list:

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

Most of instructions will act on the top frame, i.e. the head of the stack.

*States* The notion of states is defined by the following record:

```
Record state : Set := {
  sfields_f : sfields;
  heap_f : heap;
  stack_f : stack
}.
```

where `sfields` is the type for the static fields (a list of `val`, in which will be stored variables declared as static and shared by the whole program). Other fields might be added to the notion of states such as input/output buffers for APDUs.

Finally, states are used for the construction of return states of instructions. Instruction can progress normally, provoke an error or throw an exception:

```
Inductive rstate : Set :=
| Normal      : state → rstate
| Abnormal    : eLabel → state → rstate
| ThrowException : xLabel → state → rstate
```

where `eLabel` and `xLabel` record the reason of the error or the exception respectively (see Section 3.2.2).

In the definition of the module `DJVM`, the parameter state from the module type `DVM` is instantiated to `rstate` and `getstack` to the accessor field `stack_f` of the state contained in a `rstate`.

*Semantics of Instructions* Bytecode instructions update the memory of the JVM according to operands generated at compile time and to the representation of programs given below. The semantics of each instruction is formalized using a function of generic type:

$$\text{state} * \text{operands} \rightarrow \text{rstate}$$

where type `operands` is determined by the instruction to be executed.

Most instructions have a similar execution pattern:

1. the initial state is decomposed;
2. fragments of the state are retrieved;
3. observations are made to determine the new state;
4. the final state is built on the basis of the retrieved fragments and of the observations made.

To satisfy the signature of the module `DJVM`, we build a top-level execution function `exec : rstate → rstate` that expects a `Normal` state, grabs the instruction to execute and corresponding operands in the program and branches to the semantics of this instruction, or fails with an `Abnormal` state if one of the expected conditions is not met.

*Semantics of putstatic* We illustrate the modeling of instructions with the bytecode `putstatic` that store a value into the static heap.

There is four variants of `putstatic` into the JVM instruction set. One for each base type (boolean, short, int and address). The semantics presented below clutter them in a unique function by considering an extra argument (the parameter `t`) that allow to choose the right variant.

The instruction extracts from the operand stack the value `x` to be stored into the considered static field. The field is specified by its address `idx`. The type of the value is compared with the type of the field. If there

is a adequacy then the functions `res_putstatic` and `res_putstatic_ref` update the state (i.e. the considered static field is updated). Let us note that, in the case where the field is a reference, the instruction must verify the Java Card security policy. This point characterizes the difference between `res_putstatic_ref` and `res_putstatic`.

```
Definition PUTSTATIC (t : type)
  (idx : field_idx) (st : state) :=
match stack_f st with
| h :: lf ⇒
  (* Pop the topmost stack value *)
match pop_opstack t (opstack h) with
| Some (x, lv) ⇒
  match Nth_elt (sfields_f st) idx
  with
| Some nod ⇒
  match nod, t with
  | vRef _, Ref _ ⇒
    res_putstatic_ref lv st x idx
  | vPrim (vByte _), Prim Byte
  | vPrim (vBoolean _), Prim Byte
  | vPrim (vShort _), Prim Short
  | vPrim (vInt _), Prim Int ⇒
    res_putstatic lv st x idx
  | _, _ ⇒ AbortCode type_error st
  end
| None ⇒ AbortCode sfields_error st
end
| None ⇒ AbortCode opstack_error st
end
| _ ⇒ AbortCode state_error st
end.
```

where `pop_opstack` removes the first element on the stack and `res_putstatic` and `res_putstatic_ref` build the resulting state.

### 3.2.2 Errors and Exceptions Management

During the execution of a program, the normal progress of the computing might be interrupted by an ill-formed program or a non-realized condition. In the former, it corresponds to an error of the program (a `pop` on an empty stack, a value with an unexpected type, ...) that leads to the abrupt termination of the execution and that can be detected by a bytecode verifier. In the latter, the exceptions mechanism changes the normal control flow of the program and allows a efficient handling of dynamic errors of the execution.

*Representing Errors* Errors are signaled with the functions `AbortCode`, `AbortCap` and `AbortMemory` for code error, inconsistency of the program or inconsistency of the memory respectively. In turn, these functions build an abnormal state (`Abnormal`) from the reason and state given as an argument. The reason, of the following type `eReason`, is only used to determine more precisely the cause of the error:

```
Inductive eReason : Set :=
| heap_error      : eReason
| opstack_error   : eReason
```

```

| checkcast_error : eReason
| type_error      : eReason
| [...]

```

The execution can not be recovered from an abnormal state and the virtual machine terminates<sup>3</sup>.

The parameter `err_frame` of the module `DJCV` is thus instantiated to report Abnormal states.

*Representing Exceptions* In Java, when an exception is raised, a Java object, whose type gives the reason of the error, is created. Exceptions can be raised either directly by the virtual machine (in case of a division by zero for instance) or by the programmer (using the `athrow` bytecode, corresponding to the Java keyword `throw` in the source code).

When the virtual machine raises an exception, the return state of the current executed instruction has for constructor `ThrowException` and for arguments the current state and a term of type `xLabel` determining the exception to throw. A Java object of the corresponding expected type is then created in the operand stack and the function `CatchException` (that finds the exception handler as described in Sun specifications and gives the control flow to this handler or aborts the program if not found) is called.

In the case of the bytecode `athrow`, the object representing the exception to throw (that must be a subclass of the class `java.lang.Exceptions`) has already been created by previous bytecodes and the function `CatchException` can be called directly.

### 3.3 Abstractions of the Virtual Machine

In this subsection, we flesh out the construction of an abstract and offensive virtual machine from the defensive machine described in the previous subsection.

We will use the prefix `d` (resp. `a`) for the definitions of the defensive (resp. abstract) virtual machine wherever the definitions of several virtual machines are in the same scope.

#### 3.3.1 The Abstract Virtual Machine

In this subsection, we flesh out the construction of an abstract virtual machine from the defensive machine described above. This abstract machine only works with types. The execution in this context consists then in verifying the type adequacy.

*Runtime Environment* To start with, we remove value information from typed values. However, a special care has to be handled for subroutines with values representing a return address. In this case, the numerical value is required in order to determine the next instruction to execute. The types `val_prim`, `val_ref` and `val` become:

```

Inductive val_prim : Set :=
| vReturnAddress : bytecode_idx → val_prim
| vVoid : val_prim
| vBoolean : val_prim
| vByte : val_prim
| vShort : val_prim
| vInt : val_prim.

```

```

Mutual Inductive val : Set :=
| vPrim : val_prim → valu
| vRef : val_ref → valu
with val_ref : Set :=
| vRef_null : val_ref
| vRef_array : val → val_ref
| vRef_instance : class_idx → val_ref
| vRef_interface : interf_idx → val_ref.

```

Since numerical values disappear, numerical computations over elements of type `val` are omitted from the instruction semantics. For instance, the abstract version of `add` only checks if the operand stack contains at its top the compatible values (types in this case) and replaces them by the adequate type.

As a consequence of such transformations the following items from the state become redundant or useless:

- the heap, due to the absence of values (only the type of the object becomes relevant); creating an object consists then in pushing its type into the operand stack.
- the static heap, since types of values from static heap do not change during the execution; the field `static_heap_type` (that determines types for static values) is sufficient for the execution.
- the stack, due to an execution on a *method per method* basis; the stack is substituted by the topmost frame and instead of a creation of a new frame, the return type of invoked method is pushed into the operand stack and the execution continues to the next instruction of the current method.

Hence, abstract state is reduced to the notion of abstract frame:

**Definition** `astate := aframe`.

and the parameter `state` of the module `AJCV` is also instantiated to `astate`.

The trivial abstraction function between defensive and abstract values is extended to a function `alpha_da` between defensive and abstract states by abstracting values appearing in defensive states and removing useless components. `alpha_da` is in turn used to define the abstraction function  $\alpha$  of Section 2.5. Also, the BCV framework requires the definition of an order on abstract states. To this end, we follow the usual order on Java types (that we define as an inductive predicate on types) induced by class inheritance. We extend this order to list of values (operand stack and local variables) with a pointwise order and to abstract states, requiring the same program point for two comparable states.

Several instructions, for instance branching bytecodes (such as `ifnull` or `tableswitch`) have the characteristic

<sup>3</sup> Remind that the `JCV` is not threaded

to lead the execution to different program points according to values comparisons. Since numerical values are lost, the abstract version of these instructions becomes non-deterministic and their semantics must consider all the possible cases. Thus all possible return states are collected into a list and the return type of the function `exec` becomes `(list astate)`.

*Equivalence of Abstract and Defensive Machines* As required to ensure the correctness of the BCV for the JCVM, we establish execution properties stated in Lemma 1. Corresponding proofs are done in Coq by considering each bytecode of the virtual machine. For the `putstatic` bytecode, it leads to the following statement:

**Lemma** `PUTSTATIC_commut` :  $\forall$  (`st` : `dstate`)  
(`t` : `type`) (`idx` : `field_idx`),  
(`alpha_da_rs` (`dexec` (`putstatic` `t` `idx`) `st`)) =  
(`aexec` (`putstatic` `t` `idx`) (`alpha_da` `st`)).

For other bytecodes, we might have  $\leq_{\text{state}}$  instead of the equality.

The proof is obtained by case analysis on the definition of the defensive `dPUTSTATIC`. To each case analysis on this definition corresponds a refutation step thanks to a contradiction with the hypothesis or a rewriting step on the offensive version. For instance, we perform a case analysis on `(dpop_opstack t (dopstack h))` and use the following lemmas to perform a rewriting step on `dPUTSTATIC`:

**Lemma** `pop_opstack_val` :  $\forall$  (`t` : `type`) (`l` : (`list dval`)) (`v` : `dval`)  
(`l'` : (`list dval`)), (`dpop_opstack` `t` `l`) = (`Some` (`v`, `l'`))  $\rightarrow$   
(`apop_opstack` `t` (`map` `alpha_do_val` `l`)) = (`Some` (`(alpha_da_val` `v`), (`map` `alpha_da_val` `l'`))).

**Lemma** `pop_opstack_err` :  $\forall$  (`t` : `type`) (`l` : (`list dval`)),  
(`dpop_opstack` `t` `l`) = `None`  $\rightarrow$   
(`apop_opstack` `t` (`map` `alpha_da_val` `l`)) = `None`.

At the end of these steps, we obtain, as expected, two equal states.

*Monotonicity Of The Abstract Execution* The other global invariant of the execution, leading to consider each individual bytecode, concerns the monotonicity (see Section 2.4). For the `putstatic` bytecode, it leads to the following statement:

**Lemma** `exec_mon_putstatic` :  $\forall$  (`s1,s2` : `astate`)  
(`t` : `type`) (`idx` : `field_idx`),  
(`s1`  $\leq_{\text{state}}$  `s2`)  $\rightarrow$   
(`aexec` (`putstatic` `t` `idx`) `s1`)  $\leq_{\text{state}}$   
(`aexec` (`putstatic` `t` `idx`) `s2`)).

The proof is similar to the proof of correctness between defensive and abstract virtual machines. It is performed by case analysis on the definition of `aPUTSTATIC` applied on `s1`. To each case analysis, the premise is used

to deduce a corresponding rewriting step on the definition of `aPUTSTATIC` applied on `s2`. For instance, we perform a case analysis on `(apop_opstack t (aopstack s1))` and use the following property to perform a rewriting step on `s2`:

$$\forall (l,l1,l2:(\text{list } \text{aval})) (t,t1:\text{aval}),$$

$$(l1 \leq_{\text{list\_aval}} l2) \rightarrow$$

$$(\text{apop\_opstack } t1 \ l1) = (\text{Some } (t,l)) \rightarrow$$

$$(\exists (t',l'),$$

$$(\text{apop\_opstack } t1 \ l2) = (\text{Some } (t',l')) \wedge$$

$$(t \leq_{\text{aval}} t') \wedge (l \leq_{\text{list\_aval}} l'))).$$

At the end of these steps, we obtain, as expected, two comparable states.

### 3.3.2 The Offensive Virtual Machine

The offensive virtual machine will assume that every manipulated value is well typed and we will not check during execution types of value. In this virtual machine, we remove typing information from values and set:

**Definition** `val` := `Z`.

The type `z` covers all the numerical values (arithmetical and references values) encountered in the virtual machine. This leads to a simplified memory model in which type information is omitted wherever `val` is used and other fields remain unchanged.

Cross-validation results, whose description is omitted here due to space constraints, are obtained just like commutation properties on bytecodes required by Lemma 1 for the abstract virtual machine (but in that case, intra or extra-procedural bytecodes do not need to be distinguished).

---

## 4 Conclusion

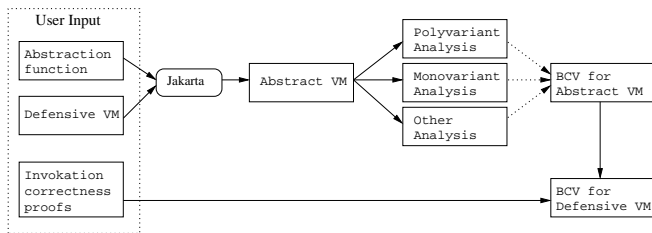
### 4.1 Summary

We have developed a general framework that establishes the correctness of a parameterized bytecode verifier, and that justifies the compositional techniques of bytecode verification. The framework has been instantiated for specific history structures that are often considered in the literature and implementations. These instantiations provide convenient entry points to our framework, and can be used in combination with Jakarta to build and validate bytecode verifiers with a high degree of automation. As illustrated in Figure 5, such a combination requires the user to provide:

- a defensive virtual machine;
- the definition of abstraction functions, in the form of Jakarta abstraction scripts, that are used to construct the abstract virtual machine and an offensive virtual machine. Scripts may contain some proof information to carry cross-machine validation;

- a formal proof of the correctness w.r.t. bytecode verification of method invocation and exception handling, i.e. an instantiation of the module `Struct_Comp` of Section 2.5;
- an instantiation of the history modules to the abstract virtual machine generated by Jakarta;

and returns an offensive virtual machine, several bytecode verifiers, and a proof that these bytecode verifiers are correct, in the sense that they will reject programs that go wrong on the defensive virtual machine, and that the offensive and defensive virtual machines coincide on programs that are accepted by bytecode verification.



**Fig. 5** Framework architecture

Such a combination has been used to good purpose for validating the Java Card platform. Using Jakarta, we have generated from the Java Card defensive virtual machine (10,000 lines of code), both the abstract and offensive virtual machine (5,000 lines of code each), as well as more than 10,000 lines of proof scripts that establish cross-machine validation and the monotonicity of the generated abstract virtual machine. We have provided another 1,500 lines of proof scripts which concern the correctness w.r.t. bytecode verification of method invocation and exception handling. Together with the output of Jakarta, these 1,500 lines provide all relevant information for the bytecode verifier to be proved correct—without any further user interaction. The bytecode verifier framework itself is constituted of 6,500 lines of proofs or specifications.

## 4.2 Related work

There is a considerable body of machine-checked specifications of execution platforms such as the JVM or .NET, many of which use the methodology instrumented in our work, see e.g. [1, 3, 7, 17]. For lack of space, we can only single out a few works:

- *Bali*: one of the most comprehensive achievements to date is that of the Bali project, which has formalized in Isabelle/HOL a large body of the Java platform, including (1) the type system and the operational semantics of the source language, with a proof of type safety; (2) the Java Virtual Machine, bytecode verifier, and lightweight bytecode verifiers, with a proof

of correctness of bytecode verification; (3) a compiler, with a proof that it preserves the operational semantics and typability, see e.g. [20, 23, 15, 16, 19]. The main differences with our work are that they do not follow our methodology (based on the defensive virtual machine) and that they do not model security features of the Java platform (whereas our formalization deals with the Java Card firewall).

- *J Book*: in their work [22], Börger, Schmid, and Stärck provide formal specifications of the offensive and defensive JVMs, and discuss their relationship, and in particular the derivation of a defensive JVM from an offensive one, but the discussion remains informal;
- *.NET*: There are also machine-checked proofs of type soundness for .NET [12, 26]. This work is more closely related to ours in the sense that [26] explicitly aims at developing tools to automate type soundness proofs. The major difference with our work is that they do not pursue cross-machine validation, and opt instead for a standard type soundness proof.

## 4.3 Future work

Our work can be pursued in three main directions: improving the framework, instantiating the framework on other case studies, and completing our models of the Java Card platform.

### 4.3.1 Enhancement of the framework

The dataflow analysis upon which our framework relies could be optimized through the use of bitmaps that record program points upon which the iteration must take place—currently we iterate abstract execution over all program points. Such optimized versions of Kildall algorithm are standard in the literature and could be added to our framework.

Further, we could provide a detailed treatment of lightweight bytecode verification algorithms, which rely on annotations at specific program points to perform bytecode verification in one pass. In fact, we have already developed an abstract lightweight bytecode verifier, and shown its soundness and completeness w.r.t. standard bytecode verification. The formalization and proof are short and crisp, but we assume that each program point comes with type information, which is not required, and departs from implementations of lightweight bytecode verification in which only junction points are annotated. It seems interesting to adapt our work so as to provide certified and realistic lightweight bytecode verifiers.

### 4.3.2 Instantiation of the framework

One main line of research for future work is to instantiate our framework to enhanced bytecode verifiers that guarantee a stronger security of applications. Indeed, there

have been many proposals of type systems for the JVM that provide stronger guarantees with respect to safety and security, and it would be interesting to adapt our virtual machine specifications to such type systems.

In fact, we have begun applying our framework to non-interference, a high-level property that assumes that executing a program will not leak confidential information. More concretely, we have built and validated a bytecode verifier for information flow for a representative fragment of the JVM, using the type system of [5]. While we use the framework described in this article, it should be noted however that, in the context of non-interference, the correctness proof for the bytecode verifier does not follow immediately from the framework, see [4].

In a different line of work, it would be nice to apply our methodology to other execution platforms, such as the .NET platform.

#### 4.3.3 Modeling the Java Card platform

Our work has focused on the verification of the bytecode verifier, and left aside several important issues in the modeling of the Java Card platforms.

It would be interesting to model native methods, both to establish some of their basic properties, and to be able to execute arbitrary Java Card applets. Further, it would be interesting to integrate novel features of Java Card 2.2, including multi-channel, or even to extend our model of the JCVM to a model of the JVM. While feasible, the latter would require to account for Java features such as dynamic class loading, garbage collection and multi-threading. Even if these features have been studied and even sometimes formally modeled, their integration within our formalization would require a substantial effort.

In another line of work, it seems very important from a practical perspective to model card management mechanisms since they have a direct impact on security. We intend to use our formalization as a basis for developing a formal model of Global Platform APIs, building up on [6].

---

## References

1. J. Andronick, B. Chetali, and O. Ly. Using Coq to Verify Java Card Applet Isolation Properties. In D. Basin and B. Wolff, editors, *Proceedings of TPHOLs'03*, volume 2758 of *Lecture Notes in Computer Science*, pages 335 – 351. Springer-Verlag, 2003.
2. G. Barthe, P. Courtieu, G. Dufay, and S. Melo de Sousa. Jakarta: tool-assisted specification and verification of the JavaCard Platform. *Journal of Automated Reasoning*, 2006. To appear.
3. G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of *Lecture Notes in Computer Science*, pages 302–319. Springer-Verlag, 2001.
4. G. Barthe and F. Kammüller. Certified bytecode verifier for non-interference. Manuscript, 2005.
5. G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich, editor, *Proceedings of TLDI'05*, pages 103–112. ACM Press, 2005.
6. S. Zanella Béguelin. Formalisation and Verification of the GlobalPlatform Card Specification Using the B Method. Manuscript, 2005.
7. G. Betarte, B. Chetali, E. Giménez, C. Loiseaux, and O. Ly. Formal Modeling and Verification of the Java Card Security Architecture: from Static Checkings to Embedded Applet Execution. In *Proceedings of ESMART'02*, 2002.
8. J. Chrzaszcz. Implementing Modules in the Coq System. In D. Basin and B. Wolff, editors, *Proceedings of TPHOLs'03*, volume 2758 of *Lecture Notes in Computer Science*, pages 270 – 286. Springer-Verlag, 2003.
9. A. Coglio. Simple verification technique for complex Java bytecode subroutines. *Concurrency and Computation: Practice and Experience*, 16(7):647–670, 2004.
10. Coq Development Team. *The Coq Proof Assistant User's Guide. Version 8.0*, January 2004.
11. S. N. Freund and J. C. Mitchell. A Type System for the Java Bytecode Language and Verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, December 2003.
12. A.D. Gordon and D. Syme. Typing a multi-language intermediate code. In *Proceedings of POPL'01*, pages 248–260. ACM Press, 2001.
13. L. Henrio and B. Serpette. A parameterized polyvariant bytecode verifier. In J.-C. Filliatre, editor, *Proceedings of JFLA'03*, 2003.
14. G. A. Kildall. A unified approach to global program optimization. In *Proceedings of POPL'73*, pages 194–206. ACM Press, 1973.
15. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, April 2002.
16. G. Klein and M. Strecker. Verified Bytecode Verification and Type-Certifying Compilation. *Journal of Logic and Algebraic Programming*, 58:27–60, 2004.
17. J.-L. Lanet and A. Requet. Formal Proof of Smart Card Applets Correctness. In J.-J. Quisquater and B. Schneier, editors, *Proceedings of CARDIS'98*, volume 1820 of *Lecture Notes in Computer Science*, pages 85–97. Springer-Verlag, 1998.
18. X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, December 2003.
19. T. Nipkow. Verified Bytecode Verifiers. In F. Honsell and M. Miculan, editors, *Proceedings of FOSSACS'01*, volume 2030 of *Lecture Notes in Computer Science*, pages 347–363. Springer-Verlag, 2001.
20. T. Nipkow and D. von Oheimb. Java<sub>light</sub> is type-safe—definitely. In *Proceedings of POPL'98*, pages 161–170. ACM Press, 1998.
21. E. Rose and K. H. Rose. Lightweight bytecode verification. In *Workshop “Formal Underpinnings of the Java Paradigm”*, OOPSLA'98, October 1998.
22. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer-Verlag, 2001.
23. M. Strecker. Formal Verification of a Java Compiler in Isabelle. In A. Voronkov, editor, *Proceedings of CADE'02*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2002.
24. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1 Runtime Environment (JCRE) Specification*, 1999.
25. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1 Virtual Machine Specification*, 1999.
26. D. Syme and A. D. Gordon. Automating type soundness proofs via decision procedures and guided reductions. In M. Baaz and A. Voronkov, editors, *Proceedings of LPAR'02*, volume 2514 of *Lecture Notes in Computer Science*, pages 418–434, 2002.

---

## Coq Formalism

The Coq system [10] is a general purpose proof assistant for the specification and verification of mathematics (in a broad sense). It is based on the Calculus of Inductive Constructions and, through to the Curry-Howard isomorphism, integrates a very rich specification language and a higher-order predicate logic.

We continue with some notations for the current Coq version, numbered 8.0. In Coq, the type of propositions is `Prop`, and the type of data is `Set`. We use  $A*B$  to denote the cartesian product of two types  $A$  and  $B$ ,  $(a,b)$  to denote pairs, `fun (x:A) => b` to denote a  $\lambda$ -abstraction,  $\forall (x:A), B$  to denote a dependent function space and  $A \rightarrow B$  to denote a non-dependent function space. An inductive type is declared with the keyword `Inductive`, its name, possibly some parameters, its type and a list of its constructors with their names. For instance, two classical examples are the type of integer and the type of polymorphic lists:

```
Inductive nat : Set :=
| 0 : nat
| S : nat → nat.
```

```
Inductive list (A : Set) : Set :=
| nil : (list A)
| cons : A → (list A) → (list A).
```

The syntax `a :: l` is a shorthand for `cons a l` and `l ++ l'` for the concatenation of the list `l` and `l'`.

A record type is declared with the keyword `Record` followed by its name, its type, and a description (name and type) of its fields. It is represented internally as an inductive type with a single constructor. Selectors are functions (defined by case-analysis) so we write `(f r)` instead of the more standard `r.f`.

Definitions are introduced by the `Definition` keyword and pattern matching over an inductive type is introduced by the `match <expr> with` notation as shown in the following intuitive example:

```
Definition is_zero (n : nat) : bool :=
match n with
| 0 => true
| (S p) => false
end.
```

As is common in dependent type theory, the specification language enforces all functions to be total. Nevertheless, partial functions can be represented in Coq for instance as done in this paper by using the lift monad which is introduced through the inductive type:

```
Inductive option (A : Set) : Set :=
| Some : A → (option A)
| None : (option A).
```

For instance, the function that computes the head of a list is defined using `option` since the head of an empty list does not exist:

```
Definition head (A : Set) (l : list A) :
(option A) :=
match l with
| nil => (None A)
| x :: _ => (Some A x)
end.
```

*Modules* Our work makes an extensive use of the interactive ML-style modules that were recently integrated to Coq [8]. Hence we briefly review the syntax for modules. The keyword

`Module Type` introduces the declaration of a type of a module, and is followed by its name, a collection a `Parameter` and `Axiom` declarations giving its signature, and it is closed by the keyword `End`. A module type can also include (and, in a certain sense, extend) other module types with the keyword `Declare Module`. A module type is implemented using the keyword `Module` (the module type it satisfies is specified after the notation `<:>`). As usual, the module must fulfill the signature of the module type it implements. Note that other modules can be given as parameters of a module, which is then called a module functor. Finally, constructions of a module can be accessed outside the module using the dot notation of qualified names or directly with the keyword `Import` followed by the module name.