ELSEVIER

Science of Computer Programming 46 (2003) 283–306

Science of Computer Programming

www.elsevier.com/locate/scico

A B model for ensuring soundness of a large subset of the Java Card virtual machine

Antoine Requet

Gemplus Research Laboratory, Av du Pic de Bertagne, 13881 Gémenos cedex BP 100, France

Abstract

Java Cards are a new generation of smart cards that use the Java programming language. As smart cards are usually used to supply security to an information system, security requirements are very strong. The byte code interpreter and verifier are crucial components of such cards, and proving their safety can become a competitive advantage. Previous works have been done on methodology for proving the soundness of the byte code interpreter and verifier using the B method. It refines an abstract defensive interpreter into a byte code verifier and a byte code interpreter. However, this work had only been tested on a very small subset of the Java Card instruction set. This paper presents a work aiming at verifying the scalability of this previous work. The original instruction subset of about 10 instructions has been extended to a larger subset of more than one hundred instructions, and the additional cost of the proof has been managed by modifying the specification in order to group opcodes by properties.

Keywords: B method; Java card; Formal specification

1. Introduction

A smart card is a small embedded system generally used to supply security to an information system. Traditionally, the application and the operating system were developed in a secure environment by the card issuer. For few years, platforms (e.g., Java Card, MultOS and Smart Card for Windows) have provided new facilities for application developers. They allow dynamic storage and execution of downloaded

E-mail address: antoine.requet@gemplus.com (A. Requet).

0167-6423/03/$ - see front matter
PII: S0167-6423(02)00095-3

executable content. Those platforms are based on a virtual machine both for portability across multiple smart card micro-controllers and for security reasons. Such architecture introduces new risks: the most important one is the possibility to attack the card from an applet by exploiting some implementation faults. In order to avoid such a risk, card manufacturers have a fairly extensive qualification process. Quality insurance requirements for smart cards are very strong. To convince the customer that the system is secure enough, card manufacturers propose to evaluate their system through a certification process.

This certification is a means for the card issuer to promote its products against its competitors. Sometimes the customer or the targeted market requires the certification. For example, the German market requires each product that uses an electronic signature to be certified at the E4 level of the ITSEC scheme. According to the certification rule and the requested level, the card issuer must provide all the elements needed by the authority to guarantee the quality of the development process. At some high levels, it is required to use formal methods and to provide the proof that security mechanisms satisfy the security policy. One of the trickiest problems is to prove the coherence of the different security mechanisms of the system. Since there are strong size constraints on the chip, the amount of memory is small. This leads Java Card to modify the security scheme. It becomes more crucial to be able to prove the correctness of the whole system security.

After a brief presentation of the Java Card security mechanisms, we sum up the state-of-the art on the formal verification of the Java byte code semantics. We then introduce the B method and emphasise the proof of the static and dynamic semantics coherence using our approach. Then, we conclude with the extension of our work and its integration in the whole Java Card model.

2. Security of the Java Card

The Java Card 2.1 standard [19] defines the CAP file (Converted APplet) i.e., the structure of the input files. For each byte code, the standard defines the conditions required for a correct execution, but not the way to ensure that those conditions are met. The Java Card virtual machine is specially designed for smart card; several features have been removed, compared to the Java virtual machine, while other features have been added (e.g., the applet firewall). The Java Card API is a set of tools or services aimed to help programmers designing Java Card applets. Due to the limited resources of the smart card (CPU, memories.), most of the tests (the verifier and part of the loader) must be done statically, outside the card. A secure link mechanism allows the card to check the integrity of the cap file; i.e., after having verified the signature, the card can safely assume that the downloaded program has the required properties, and that a valid verifier has checked it. Of course the certificate can only be provided by a trusted third-party authority.

In fact, the security provisions are scattered across different components: a verifier, a converter, an on-card loader, a firewall and an interpreter (see Fig. 1). Moreover a specific applet is used to manage the applets: the Java Card runtime environment

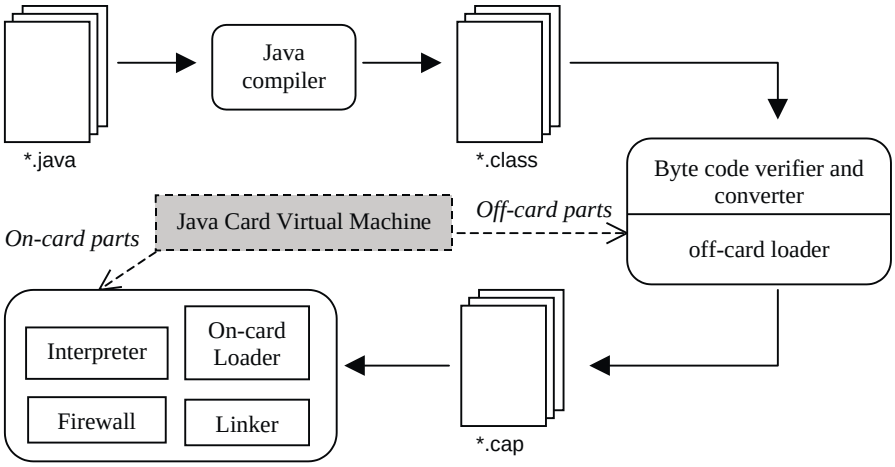

Fig. 1. Java Card environment.

(JCRE). It is used to select and deselect applets, and also contain the registers of the selected applets and of the currently active applet.

The byte code verifier performs a static analysis of Java class files in order to verify some basic security properties. The main property verified is type correctness, but other additional checks are performed: for example it is checked that the maximum runtime stack size correspond to the declared one. Performing those tests allows suppressing the corresponding runtime tests from the interpreter. This improves the interpreter performances, and reduces its memory usage.

While the virtual machine ensures Java language-level security, the firewall performs additional runtime checks. This mechanism is in charge of the applet isolation and of the control of object accesses. For example, it prevents unauthorised accesses to the fields and the methods of class instances. An applet may share objects with other applets, so the applet firewall must control the access to the shareable interface of these objects. This component is of prime importance for the system security.

The security policy has to express the correct confinement of the applets and the correct access to shared objects. Respecting the typing rules associated to the access rules of the firewall guarantees this security policy. Thus, we have to verify that the elements performing those checks are correctly implemented and that they are consistent. A formal specification of these mechanisms must be done even if the formal proof is costly. Several elements have already been modelled: the verifier [4] and partially the JCRE with an emphasis on the firewall [13]. We present here a method guaranteeing that the consistency between the byte code verifier and the interpreter.

3. Related work

There has been much work on a formal treatment of Java but no work has been done in order to formally verify whether a given security policy is correctly implemented

by a virtual machine. All the works on Java and the Java byte code focus on a formal definition of the semantics. At the Java language level [14,20] define a formal semantics for a subset of Java in order to prove the soundness of its type system. Qian [17] considers a subset of the byte code and aims at proving the runtime correctness from its static typing. Then, he proposes the proof of a verifier that can be deducted from the virtual machine specification.

An interesting work has been done by Cohen [5]. He proposes a formal implementation of a defensive virtual machine. It is possible to prove that his model is equivalent to an offensive interpreter plus a sound byte code verifier. Posegga and Vogt [15] propose a verification mechanism based on a model checker. They have shown the easiness of the proof process using the SMV tool. Goldberg [9] proposes a formal specification of the byte code verifier for the data flows analysis. His approach is close to the implementation but he simplifies the problem by neglecting to check subroutines. In the Bali project [16], Push proves a part of the Java Virtual Machine using the prover Isabelle/HOL. In [17], Qian gives a specification of the byte code verifier and then proves its correctness.

Other works are more specifically targeted at Java Card. Using the B method [2], Lanet and Requet [11] proves the correctness of byte code optimisations performed by the Java Card applet converter, by specifying the optimised byte code as a refinement of the original code. A complementary work has been done by Denney and Jensen [6]. They prove the correctness of the Java Card conversion using the Coq proof assistant.

Java Card is not the only environment that has been subject to formal studies. The first formal treatment of a smart card operating system has been published in [3]. The interpreter of the windows for smart card runtime environment has been modelled in [10]. The authors uses abstract state machines to describe the operational semantic of the interpreter and proves that a program will not read or write to illegal memory locations. A similar property is proved in [18], where the author describes how they used Z to prove segregation between applets in a smart card operating system.

4. The approach used

The main purpose of our approach is to ensure the soundness of the type system. Principles described in [4] are used to formally specify the Java byte code interpreter. The main idea is to use a formal description of an abstract defensive byte code interpreter operating on types. This abstract interpreter defines the checks needed to ensure a type safe byte code execution and defines the expected security policy. It acts as a gluing specification ensuring consistency between the interpreter specification and the specification of the verifier (Fig. 2).

The runtime checks performed by the defensive interpreter are then removed and converted to static constraints on the byte code during the refinement process. During this process, the proof obligations of the refinement ensure the validity of the static constraints specified.

At the last refinement step, the machine is used to merge a byte code verifier, which enforces the static constraints, and an aggressive interpreter, corresponding to

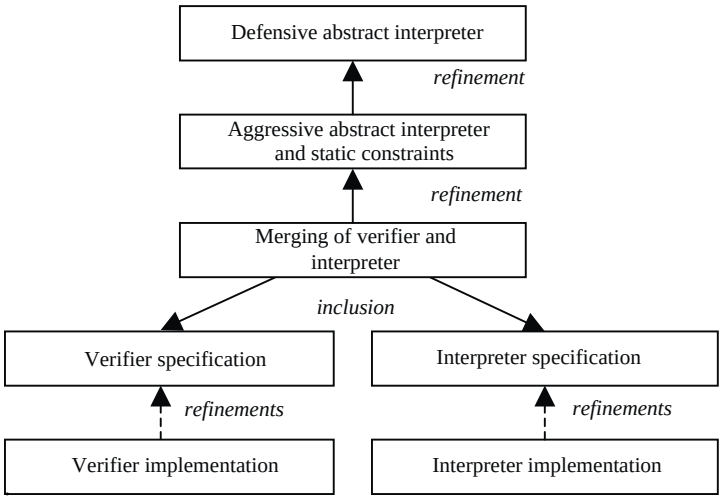

Fig. 2. Overview of the approach.

the implementation of the Java Card virtual machine. The refinement proofs ensure that the properties defined in the abstract interpreter are preserved by the aggressive one.

This approach ensures the soundness of the byte code verifier and the interpreter. That is, the byte code interpreter relies only on tests that are performed. Moreover, from the verifier point of view, this proves that the properties verified are enough to guarantee a safe byte code execution: a property that would not have been verified would generate proof obligations impossible to prove in the verifier part. Lastly, generating the code for the interpreter and the verifier should provide strong assurance in the correctness of the implementation.

Initially, a small instruction set composed of about ten instructions and a simplified lattice has been used. This approach was suitable for this small instruction set, but extending it to the whole Java Card instruction set did not scale well. Surprisingly, the main scalability problem was not the increased complexity implied by new features of the virtual machine, but the size of the specification and the amount of similar manual proofs required.

More exactly, each instruction needed several manual proofs and both the response time and memory requirement of the prover were too large to completely demonstrate the proofs. The next part of this paper focuses on describing how the approach has been extended for a large subset of the Java Card virtual machine.

5. Machine considered

5.1. Instruction set

The Java Card subset considered consists of all the stack manipulation instructions, most of the control flow instructions and instructions manipulating local variables.

As the aim of this work was to verify the scalability of the approach, instructions that would drastically increase the complexity of the model have been left out. Especially, those instructions include the instructions used for subroutines, for method calls and for objects handling. The difficulties implied by those instructions have already been widely studied [1,7,8], and there are known solutions for handling them. Moreover, those difficulties usually involve few instructions, and are not subject to the same scalability problems. The handling of exceptions and subroutines will be added later, when the scalability of the model will be resolved. We will use a model developed as an extension of [11] based on [1] and very close to [8]. Such an extension will however require adding more information than is currently available (for example, subroutines labelling).

Other points, such as subtyping can be added in a straightforward manner, by modifying the definitions used to compare types.

So, the chosen instruction set is neither representative of the full Java Card instruction set nor representative of the tricky parts of the full instruction set. However, it appears as a valid choice for studying the problems that are encountered when extending a 10-instruction subset to the full instruction set.

In order to simplify the specification and the proof process, the opcodes are grouped by properties. Sets are defined to contain opcodes with similar properties. For example, the following sets are used:

- *OP_NEXT*: This set contains opcodes that can go to the next instruction after execution. This includes nearly all the instruction, except for unconditional jumps.
- *OP_BRANCH* and *OP_BRANCH_W*: the set of opcodes that may perform a relative branch, where the target is defined by the first parameter. There are two sets, since the branch can be defined by a signed byte parameter (*OP_BRANCH*) or a signed short parameter (*OP_BRANCH_W*).
- *OP_NEXT_FRAME_READ*: the set of opcodes reading a value from the local variables.

A given opcode can be part of several sets. For example, instructions that perform conditional branch are both elements of *OP_NEXT* and *OP_BRANCH*. Although every Java Card opcodes cannot fit in a group, such a grouping scheme highly simplifies the specification. The opcodes that do not fit into groups have to be handled as special cases.

A subset of instructions manipulating the stack is created. Each of those instructions behaviour is modelled as first removing elements from the stack, and adding new elements to the resulting stack. For example, the instruction *iadd*, which adds the two topmost elements of the stack together, and replaces them by the result, is considered as being an instruction that pops two integers from the stack and pushes an integer.

To represent this, two constants have been added: *tpushed* and *tpopped*. Those constants are defined as partial maps from opcodes to sequence of types. *tpopped* defines the types that are expected to be removed from the top of the stack, and *tpushed* defines the types to be pushed onto the stack when the instruction is executed. In the

$$
\begin{array}{l}
max\_locals \in \mathbf{NAT} \wedge \\
max\_stack \in \mathbf{NAT} \wedge \\
method \in \mathbf{seq1}(BYTE) \wedge \\
opcode\_locations \subseteq \mathbf{dom}(method)
\end{array}
$$

Fig. 3. Constants used to represent a method.

previous example, *tpushed*(*iadd*) is equal to the one element sequence [*integer*], and *tpopped*(*iadd*) is equal to the sequence [*integer*, *integer*].

One drawback of the grouping scheme is that it generates more complicated proof obligations that require increased manual interaction. However, an advantage is that those proof obligations are more generic and can usually be used to discharge nearly all the proof obligations corresponding to the opcodes within the group.

5.2. State of the machine

We consider the execution of one method. This is enough to verify the consistency between the interpreter and the verifier. Thus the verification can be performed a method at a time, provided that some information about the global context is accessible. This information corresponds to the content of the class file's constant pool.

An abstract set $BYTE$ is defined, the method being considered as a sequence of $BYTE$. Since its contents do not change during the interpretation, it is defined as a constant. Method is assigned the B type **seq1**($BYTE$), that correspond to nonempty sequences of bytes. This low-level representation allows treating cases where the method is not syntactically correct. For example cases where the method contains invalid instructions, or jumps into the middle of an instruction.

Some additional information on the method is added: *max_stack* corresponds to the maximum size of the local stack during the execution of the method, and *max_local* to the maximum number of local variables used. Lastly, the set *opcode_locations* corresponds to the set of valid addresses within the method. As this last information is not directly available within the Java class file, it has to be computed before the method is executed. B sequences are functions, allowing specifying that *opcode_locations* is a subset of the domain of the sequence *method*. Fig. 3 provides the corresponding fragment of the B specification.

For the most abstract specification, we are only interested in the types contained in the stack and the frame. So, the state consists of:

- the program counter, which points to the instruction currently being executed,
- the typing of the runtime stack,
- the typing of the frame.

This state is defined by the variables shown in Fig. 4. For now, the variable *frame_type* contains the content of the frame, and is defined as a partial map from integer to type (more exactly, from the interval 0 to the maximum variable number

$$
\begin{array}{l}
frame\_type \in 0..max\_locals\text{-}1 \mapsto TYPE \wedge \\
stack\_type \in \mathbf{seq}(TYPE) \wedge \\
\mathbf{size}(stack\_type) \leqslant max\_stack \wedge \\
apc \in opcode\_locations
\end{array}
$$

Fig. 4. Variables representing the state of the machine.

$$
\begin{array}{l}
BYTE\_to\_OPCODE \in BYTE \rightarrow OPCODE \wedge \\
BYTE\_to\_signed \in BYTE \rightarrow \mathbf{INT} \wedge \\
BYTE2\_to\_signed : (BYTE \times BYTE) \rightarrow \mathbf{INT}
\end{array}
$$

Fig. 5. Functions handling byte conversions.

to type). The variable *stack_type* represents the content of the stack, and is defined as a sequence of types. *apc* is defined as being a value in *opcode_locations*, always ensuring the applet confinement. An additional invariant ensures that the stack never overflows.

Since we manipulate byte, and not more abstract data types, we need some functions converting bytes to opcodes or values. Fig. 5 lists some of the B functions defined. The functions *BYTE_to_signed* and *BYTE2_to_signed* allow converting a byte or a short into a signed value useable within the specification. Those functions are defined as constants, and are used to get the opcodes and the parameters from the method.

6. The defensive interpreter

The defensive interpreter performs an abstract execution of the method, and ensures that every instruction can be executed in a safe way using runtime tests. Each Java opcode has an associated B operation describing the expected semantics.

To simplify the specification, a few more convenience definitions are introduced. They are shown in Fig. 6.

The first definition corresponds to a function returning the opcode for the specified location in the method. The second one is used to access parameters associated to opcodes. The next one computes the address of the next instruction based on the number of additional parameters of the opcode. It uses the function *parameters_size* which provides for each opcode the number of bytes used by its parameters. The last definition is a predicate ensuring that the stack can be updated according to the definition of the current opcode. That is, it ensures that the execution of the instruction will not introduce stack underflow or overflow, and that the types expected are present on top of the stack. This definition uses the B high restriction operator ($\uparrow$), which restricts a sequence to the given number of its first elements. More exactly, if s is a sequence and n is an integer less than the size of s, then $s \uparrow n$ is a list corresponding to the n first elements of s.

$$
\begin{aligned}
&opcode(pc) == BYTE\_to\_OPCODE(method(pc));\\
&parameter(pc, xx) == method(pc + xx);\\
&succ\_pc(pc) == pc + parameters\_size(opcode(pc)) + 1;\\
&can\_update\_stack(pc) == \mathbf{size}(stack\_type) \geqslant \mathbf{size}(tpopped(opcode(pc))) \wedge\\
&\qquad \mathbf{size}(stack\_type)\text{-}\mathbf{size}(tpopped(opcode(pc))) + \mathbf{size}(tpushed(opcode(pc)))\\
&\qquad\qquad \leqslant max\_stack \wedge\\
&\qquad stack\_type \uparrow \mathbf{size}(tpopped(opcode(pc))) = tpopped(opcode(pc))
\end{aligned}
$$

Fig. 6. Definitions.

```
op_iload =
SELECT
   opcode(apc)= ILOAD
THEN
   IF
      BYTE_to_unsigned(parameter(apc, 1)) ∈ 0..max_locals-1 ∧
      frame_type(BYTE_to_unsigned(parameter(apc, 1))) =
                                 frame_type_used(opcode(apc)) ∧
      succ_pc(apc) ∈ opcode_locations ∧
      can_update_stack(apc)
   THEN
      apc := succ_pc(apc)||
      stack_type := tpushed(opcode(apc))
         ^(stack_type ↓ size(tpopped(opcode(apc))))
   END
END;
```

Fig. 7. Specification of the operation corresponding to the iload opcode.

To specify the operations, we use event driven B, and associate a guard corresponding to the expected opcode of the operation. The operation will be triggered when the guard is true, that is, when the corresponding opcode is encountered.

Each operation performs tests ensuring that it can safely be executed and then updates the state of the machine. For example, the specification of the *iload* instruction, which loads an integer local variable onto the stack is given in Fig. 7. The low restriction operator $\downarrow$ is similar to the high restriction operator previously described, but is used to restrict a sequence to its last elements, and the $^\wedge$ operator is used to concatenate two sequence.

In this example, the content of the *SELECT* clause means that this operation will be triggered when an *iload* opcode is encountered within the method. Then, the tests within the *IF* clause correspond to the runtime tests performed when executing the instruction: the two first checks ensure that the local variable exists and is defined, and that the types it uses match with the expected types, ensuring correct typing. The next checks ensure the confinement of the applet execution, by testing if the program counter is still within the method body after the operation is performed. The last check tests for the stack underflow and overflow, and ensures that the types expected within

```
op_ifle =
SELECT
    opcode(apc) = IFLE
THEN
    CHOICE
        IF
            succ_pc(apc) ∈ opcode_locations ∧
            can_update_stack(apc)
        THEN
            apc := succ_pc(apc) ||
            stack_type := tpushed(opcode(apc))
                ^(stack_type ↓ size(tpopped(opcode(apc))))
        END
    OR
        IF
            apc + parameter(apc, 1) ∈ opcode_locations ∧
            can_update_stack(apc)
        THEN
            apc := apc + parameter(apc, 1) ||
            stack_type := tpushed(opcode(apc))
                ^(stack_type ↓ size(tpopped(opcode(apc))))
        END
    END
END;
```

Fig. 8. Specification of the operation corresponding to the ifle opcode.

the stack match with the types found. If all the tests are successful, the state of the interpreter is updated: the program counter is set to the next instruction, and the content of the frame is modified. The function *frame_type_used* is used to get the expected type of the local variable used, and is similar to the functions *tpopped* and *tpushed*.

As this defensive interpreter only operates on types, its specification cannot be deterministic: some instruction behaviour may depend on the values stored in the stack or within the variables. An example of this is the instructions performing conditional branch depending on stack values. As only the type of those values is known, it is not possible to decide if the branch is taken. Instead, it is specified that, either the jump is performed, or the execution continues to the next instruction.

The specification of the *ifle* instruction is given in Fig. 8. The B substitution CHOICE represents a nondeterministic choice. When the operation is called either the substitutions between CHOICE and OR, either the substitutions between OR and END are executed. However, this substitution does not define which one will be executed yet.

The first part of the clause represents the case where the execution continues to the next instruction and the second to the case where the execution continues to the branch target. Determinism will be added later within the interpreter specification, since the values stored within the stack are not available yet, and it is not necessary to know which branch will be taken for performing the verification.

$$
\begin{array}{l}
frame\_type\_s \in \mathbf{seq}(0..max\_locals\text{-}1 \mapsto TYPE) \wedge \\
stack\_type\_s \in \mathbf{seq}(\mathbf{seq}(TYPE)) \wedge \\
stack\_type\_s(apc) = stack\_type \ \wedge \\
frame\_type\_s(apc) = frame\_type
\end{array}
$$

Fig. 9. Definition of the static variables.

7. Replacement of runtime tests by static properties

The replacement of runtime tests by static properties corresponds to the first refinement of the defensive interpreter as previously described in Fig. 2. This new refinement introduces new variables that are used to express the properties, and to state whether those properties are met by the method or not. Then, the specification of the operation is updated to remove the dynamic tests.

7.1. Introduction of new variables

Expressing the static properties requires adding additional information about the method. Specifically, we need to express properties of the typing content of the stack, and of the potentially used local variables for each instruction.

This information will be computed and checked later by the verifier. The verifier can perform such a type inference, because of constraints imposed on all valid Java programs [12]. This constraint allows Java programs to be verified in a finite time. Java verifiers would reject any program where this information could not be computed.

Two new variables are introduced (Fig. 9): *stack_type_s* and *frame_type_s*, representing typing information for the stack and the frame. For each instruction of the method, they define the expected content of the stack and the frame. These variables are linked to the state of the interpreter, by stating that the current state of the interpreter must match the stack and frame content of *stack_type_s* and *frame_type_s* for the current instruction.

7.2. Definition of the static properties

We are considering three different static properties. These properties correspond to properties on the control flow (applet confinement), on the stack (correct typing and no underflow/overflow), and on the validity of local variables access (correct typing). These static properties are expressed as invariants of the machine, by predicates defining properties that must be true before the instruction is executed, so that the execution of this instruction leads to another safe state. They can be compared to preconditions that must be enforced before the program is executed.

The confinement property is expressed by defining properties that must be enforced for opcodes of different groups. The only requirements needed to be expressed for the confinement property relates to the program counter. Fig. 10 describes the invariant corresponding to the confinement property.

$$
\begin{array}{l}
\mathit{static\_flow\_checked} == \\
\forall \mathit{pc}.((\mathit{pc} \in \mathit{opcode\_locations} \wedge \mathit{opcode}(\mathit{pc}) \in \mathit{OP\_NEXT}) \\
\qquad \Rightarrow \\
\qquad \mathit{succ\_pc}(\mathit{pc}) \in \mathit{opcode\_locations}) \wedge \\
\forall \mathit{pc}.((\mathit{pc} \in \mathit{opcode\_locations} \wedge \mathit{opcode}(\mathit{pc}) \in \mathit{OP\_BRANCH}) \\
\qquad \Rightarrow \\
\qquad \mathit{pc} + \mathit{BYTE\_to\_signed}(\mathit{method}(\mathit{pc}+1)) \in \mathit{opcode\_locations}) \wedge \\
\forall \mathit{pc}.((\mathit{pc} \in \mathit{opcode\_locations} \wedge \mathit{opcode}(\mathit{pc}) \in \mathit{OP\_BRANCH\_W}) \\
\qquad \Rightarrow \\
\qquad \mathit{pc} + \mathit{BYTE2\_to\_signed}(\mathit{method}(\mathit{pc}+1), \mathit{method}(\mathit{pc}+2)) \in \mathit{opcode\_locations})
\end{array}
$$

Fig. 10. Static properties for confinement.

$$
\begin{array}{l}
\mathit{static\_stack\_checked} == \\
\ldots \\
\forall \mathit{pc}.((\mathit{pc} \in \mathit{opcode\_locations} \wedge \mathit{opcode}(\mathit{pc}) \in \mathit{OP\_BRANCH}) \\
\Rightarrow \\
\mathbf{size}(\mathit{stack\_type\_s}(\mathit{pc})) - \mathbf{size}(\mathit{tpopped}(\mathit{opcode}(\mathit{pc}))) + \mathbf{size}(\mathit{tpushed}(\mathit{opcode}(\mathit{pc}))) \\
\qquad \leqslant \mathit{max\_stack} \wedge \\
\mathbf{size}(\mathit{tpopped}(\mathit{opcode}(\mathit{pc}))) \leqslant \mathbf{size}(\mathit{stack\_type\_s}(\mathit{pc})) \wedge \\
\mathit{stack\_type\_s}(\mathit{pc}) \uparrow \mathbf{size}(\mathit{tpopped}(\mathit{opcode}(\mathit{pc}))) = \mathit{tpopped}(\mathit{opcode}(\mathit{pc})) \wedge \\
\mathit{stack\_type\_s}(\mathit{pc}+1+\mathit{BYTE\_to\_signed}(\mathit{method}(\mathit{pc}+1))) = \\
\qquad \mathit{tpushed}(\mathit{opcode}(\mathit{pc})) \frown (\mathit{stack\_type\_s}(\mathit{pc}) \downarrow \mathbf{size}(\mathit{tpopped}(\mathit{opcode}(\mathit{pc})))) \wedge \\
\ldots
\end{array}
$$

Fig. 11. Stack property for branching opcodes.

The stack properties are expressed in a similar way. They relate the content of the static typing stacks before the instruction to the content of those stacks after the instruction is executed. For example, in the case of branching opcode, it is stated that:

- the size of the stack after the execution of the instruction is less than *max_stack*,
- the stack does not underflow during the execution of the instruction,
- the resulting stack does not underflow,
- the static stack for the branch target matches the resulting stack.

The property associated to the stack for branching opcodes are given in Fig. 11.

The last set of properties ensures the consistency of the frame accesses. The corresponding definition for opcodes reading the frame and going to the next instruction is given Fig. 12. As the function *frame_type_s* is only defined for usable variables, it is stated that the next frame has to be included in the current frame, ensuring that information on variables is either unchanged, either lost.

Three boolean variables are defined: *flow_checked, stack_checked* and *frame_checked*. Those variables correspond to the result of the verifier, and are set to true only if the program has the corresponding property. Invariants are added to link those values to the static properties defined as shown in Fig. 13.

$$
\begin{array}{l}
static\_frame\_checked == \\
\forall pc.((pc \in opcode\_locations \wedge opcode(pc) \in OP\_NEXT\_FRAME\_READ) \\
\Rightarrow \\
BYTE\_to\_unsigned(method(pc+1)) \in 0..max\_locals - 1 \wedge \\
frame\_type\_s(pc)(BYTE\_to\_unsigned(method(pc+1))) \\
\qquad\qquad = frame\_type\_used(opcode(pc)) \wedge \\
frame\_type\_s(\text{succ\_pc}(apc)) \subseteq frame\_type\_s(pc))
\end{array}
$$

Fig. 12. Frame property for opcodes reading the frame.

$$
\begin{array}{l}
flow\_checked \in \mathbf{BOOL} \wedge \\
(flow\_checked = \mathbf{TRUE} \Rightarrow static\_flow\_checked) \wedge \\
\\
stack\_checked \in \mathbf{BOOL} \wedge \\
(stack\_checked = \mathbf{TRUE} \Rightarrow static\_stack\_checked) \wedge \\
\\
frame\_checked \in \mathbf{BOOL} \wedge \\
(frame\_checked = \mathbf{TRUE} \Rightarrow static\_frame\_checked)
\end{array}
$$

Fig. 13. Invariant defining static properties.

$$
\begin{array}{l}
\mathbf{op\_iload} = \\
\mathbf{SELECT} \\
\quad opcode(apc) = ILOAD \wedge \\
\quad flow\_checked = \mathbf{TRUE} \wedge stack\_checked = \mathbf{TRUE} \wedge frame\_checked = \mathbf{TRUE} \\
\mathbf{THEN} \\
\quad apc := \text{succ\_pc}(apc) \| \\
\quad stack\_type := tpushed(opcode(apc)) \frown (stack\_type \downarrow \mathbf{size}(tpopped(opcode(apc)))) \\
\mathbf{END};
\end{array}
$$

Fig. 14. Refinement of the iload operation.

The specification of the operations is nearly the same as the defensive one. The difference is that tests against the values of the checked variable are placed within the guard, and that the dynamic tests are removed. For example, the specification of the iload operation is given in Fig. 14.

The refinement mechanism ensures that every refined operation can occur only in a state corresponding to one in which the abstract operation could occur, and that the refined operation behaves as the abstract operation. So, proving that the new specification is a valid refinement of the defensive interpreter ensures the soundness of the byte code verifier and the interpreter.

The main difference between the defensive interpreter and the refined interpreter, apart the fact that no runtime tests are performed is that there is not a strict correspondence between the operations triggered by the defensive interpreter and the refined one. If the method can be checked, then the operations triggered will be the same as the abstract ones. However, if the method contains an error, the abstract operations

will be called until the program counter reach the error, but no refined operation will be called at all. Such a difference is made possible by using event driven B, which allows strengthening the guards of the operations. So, refined operations are potentially triggered less often than the abstract operations, but they have to conform to their abstract specification.

8. Inclusion of the verifier and the interpreter

This refinement corresponds to the merging of verifier and interpreter shown in Fig. 2. It is used to include both the verifier and a "real" interpreter. By real, we mean an interpreter that does not perform an abstract interpretation of the method based on the types of the values, but only uses values. At this point, most of the variables used previously are removed from this machine, and linked with invariant to the variables introduced by the verifier and the interpreter machines.

8.1. Verifier specification

The verifier specification contains only one operation, which performs the byte code verification, and returns a boolean value corresponding to the result of the verification. Since the verifier computes the content of the variables *stack_type_s* and *frame_type_s*, those variables are removed from the defensive machine refinement, and declared in the verifier machine. This introduces an implicit gluing invariant ensuring that the content of those variables still matches the content defined in the defensive machine.

A simplified specification of the *verify_method* corresponding to the previously described properties is given in Fig. 15.

The verifier machine is included in the refinement, and called during the initialisation to define the values of the variables *flow_checked*, *stack_checked* and *frame_checked* as shown in Fig. 16.

The implementation of the verifier performs the type inference using a fixpoint computation as described in [4]. The presence of embedded loops increases the difficulty of the proof process. However, the fact that this model does not include subtyping greatly simplifies the verifier. Splitting the implementation in several small operations allows the automatic prover to discharge up to 95% of the proof obligations. However, proving the remaining 5% proof obligations still remains costly.

8.2. Interpreter specification

The interpreter is defined as a machine similar to the abstract interpreter, excepted that it is an aggressive interpreter, and that it operates on values instead of types. Its state consists of a pointer to the current instruction executed (*dpc*, for dynamic program counter), the values stored in the stack (*stack_value*) and the values stored in the frame (*frame_value*).

Note that the program pointer is no longer restricted to the set *opcode_locations*: this restriction will be enforced by the gluing invariant relating the value of *apc* to the

$$
\begin{array}{l}
flow\_ok,\ stack\_ok,\ frame\_ok \leftarrow \text{verify\_method} = \\
\textbf{ANY}\ fl\_ok,\ st\_ok,\ fr\_ok\ \textbf{WHERE} \\
\quad fl\_ok \in \textbf{BOOL} \wedge st\_ok \in \textbf{BOOL} \wedge fr\_ok \in \textbf{BOOL} \wedge \\
\quad (fl\_ok = \textbf{TRUE} \Rightarrow static\_flow\_checked) \wedge \\
\quad (st\_ok = \textbf{TRUE} \Rightarrow static\_stack\_checked) \wedge \\
\quad (fr\_ok = \textbf{TRUE} \Rightarrow static\_frame\_checked) \\
\textbf{THEN} \\
\quad flow\_ok,\ stack\_ok,\ frame\_ok := fl\_ok,\ st\_ok,\ fr\_ok \\
\textbf{END}
\end{array}
$$

Fig. 15. Specification of the verify_method operation.

$$
\begin{array}{l}
\textbf{INITIALISATION} \\
\quad flow\_checked,\ stack\_checked,\ frame\_checked \leftarrow verify\_method || \\
\quad \ldots
\end{array}
$$

Fig. 16. Call of the verify_method operation.

$$
\begin{array}{l}
dpc \in \textbf{NAT} \wedge \\
stack\_value \in \textbf{seq}(\textbf{INT}) \wedge \\
frame\_value \in 0..max\_locals \rightarrow \textbf{INT}
\end{array}
$$

Fig. 17. State of the interpreter.

value of *dpc*. Removing this restriction from the interpreter abstract machine simplifies later proof in this machine (Fig. 17).

To ensure the consistency between the abstract interpreter and the concrete interpreter, we have to glue the state of the abstract interpreter to the state of the concrete interpreter using additional invariants. For the stack, it is ensured that both the stack containing the values and the stack containing the types have the same size. That is, every defined value has a type, and every type has a value. The invariant relating the types frame to the values frame is not as simple: it is stated that the domain of the typing frame has to be included within the domain of the value frame. That is, every variable that may be used is defined.

The domain *value_frame* can be larger than the domain of *type_frame*, since every local variable has a value even if its type is not defined. Last, the current instruction executed must be the same for both interpreters. Those three invariants, shown in Fig. 18 ensure that we have not specified two different and unrelated interpreters.

The guards corresponding to the operations are unchanged. However, the body of the operation now only calls the associated operation of the interpreter. For example, Fig. 19 shows the operation *op_iload* that calls the corresponding operation *int_iload* of the interpreter.

int_iload is the operation corresponding to the opcode iload within the interpreter machine (Fig. 20). It pushes the value contained in the specified local variable onto

$$
\begin{array}{l}
apc = dpc \wedge \\
\mathbf{size}(stack\_type) = \mathbf{size}(stack\_value) \wedge \\
\mathbf{dom}(frame\_type) \subseteq \mathbf{dom}(frame\_value)
\end{array}
$$

Fig. 18. Gluing invariant for the interpreter.

$$
\begin{array}{l}
\mathbf{op\_iload} = \\
\mathbf{SELECT} \\
\quad opcode(dpc) = ILOAD \wedge \\
\quad flow\_checked = \mathbf{TRUE} \wedge stack\_checked = \mathbf{TRUE} \wedge frame\_checked = \mathbf{TRUE} \\
\mathbf{THEN} \\
\quad int\_iload \\
\mathbf{END};
\end{array}
$$

Fig. 19. iload operation for the second refinement.

$$
\begin{array}{l}
\mathbf{int\_iload} = \\
\mathbf{PRE} \\
\quad \text{succ\_pc}(dpc) \in \mathbf{NAT} \wedge \\
\quad \mathbf{size}(stack\_value) < max\_stack \wedge \\
\quad BYTE\_to\_unsigned(parameter(dpc, 1)) \in \mathbf{dom}(frame\_value) \\
\mathbf{THEN} \\
\quad dpc := \text{succ\_pc}(dpc) \| \\
\quad \mathbf{LET}\ var\_value\ \mathbf{BE} \\
\quad\quad var\_value = frame\_value(BYTE\_to\_unsigned(parameter(dpc,1))) \\
\quad \mathbf{IN} \\
\quad\quad stack\_value := var\_value \rightarrow stack\_value \\
\quad \mathbf{END} \\
\mathbf{END;}
\end{array}
$$

Fig. 20. iload operation for the interpreter.

the stack. This is done using the arrow operator, which adds a value to the beginning of a sequence. As this interpreter is implemented in a separate machine that has no knowledge of the constraints enforced on the byte code, the preconditions ensuring that the execution can be performed have to be provided. Preconditions are specification substitutions that specify the conditions that have to be true when the operation is called. They are used to generate proof obligations, and to achieve the proof.

The consistency between those preconditions and the byte code verification is ensured by the proof obligations generated when the operation *int_iload* is called from the operation *op_iload*: it will be needed to prove that the content of the *op_iload* guard implies the *int_iload* precondition.

Another point is that, instead of using a different machine, the interpreter could have been treated as a refinement of the abstract defensive machine, in a way similar to what has been done in [11]. However, separating the interpreter from the abstract

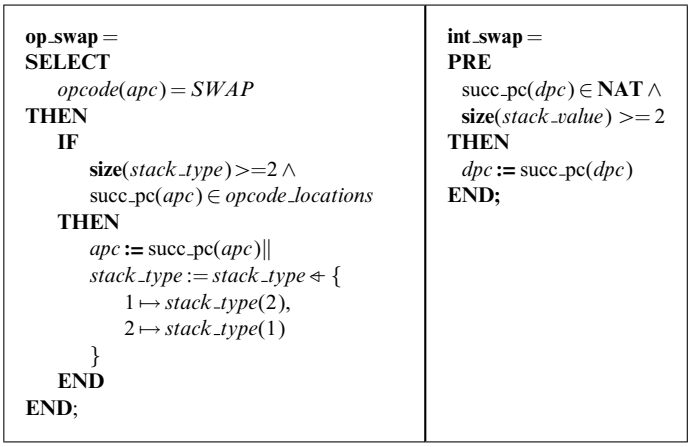

Fig. 21. Incorrect, but provable specification.

specification seems to be a better solution, since less proof obligations will be generated: proofs are needed when the interpreter is included within the refinement, but not in later refinements of the interpreter, allowing to focus on the interpreter implementation. Moreover, implementing the interpreter as distinct machines allows to clearly separate the proof of consistency from the implementation.

8.3. Limitation of the approach

A limitation of this approach corresponds to the gluing invariant between the real interpreter and the abstract interpreter. This invariant specifies that each variable of the interpreter has a type. However, although this invariant ensures that the verifier modifies existing variables, it does not ensure that the interpreter modifies the variables according to their types. Special care must be taken to check that the defensive machine really describes the typing rules within the interpreter.

Fig. 21 shows an erroneous specification that could be proved. The *op_swap* operation corresponds to the specification of the swap instruction in the defensive machine, and *int_swap* to an incorrect specification in the interpreter machine.

The *swap* instruction exchanges the two values on top of the stack using the B function-overloading operator. The specification in the defensive machine specifies that the types contained within the stack are swapped. However, the corresponding interpreter operation is incorrect and does not modify the content of the stack. In this case, there is an inconsistency between the behaviour of the abstract interpreter and the behaviour of the real interpreter that will not be detected during the proof, since the number of variables is unchanged.

A suggested extension was to specify a full typed defensive machine, that would operates both on values and types instead of an abstract defensive machine only considering types, and remove the typing information during the refinement. However,

this would incur the same risks in the abstract machine, where the specified behaviour would have to be consistent with the modifications performed on types. It would be similar to the currently used approach, but would be more complicated since the abstract specification would be larger. Such an approach may, however, reduce the risks of specification errors by placing both the typing information and the machine behaviour in the same machine.

Although we present this as a limitation of the approach, this is more a specification error. In the previous example, the specified *swap* instruction does not correspond to the Java virtual machine instruction that swaps the two topmost values on the stack. Instead, it corresponds to an unsafe type conversion instruction that exchanges the type of the two topmost values. The proof that would be performed still ensures that the interpreter complies with its specification, however the specification would not conform to the Java virtual machine specification.

9. Proof of the specification

The specification of the defensive virtual machine and its refinement is about 10 000 lines of B specification. The *Atelier B* tool, that we used for this specification generates nearly 3000 proof obligations. It should be noted; however, that the proofs are not complicated by themselves. The main difficulty lies in their number: proving the correctness of the specification corresponds to discharge a lot of simple proof obligations.

9.1. The defensive machine

The proof obligations generated for this refinement corresponds to prove the preservation of the invariant: that is ensuring that the program counter points to a valid opcode location in the method, and that the stack never overflows. It ensures, however, that no runtime test possibly breaking the invariant has been forgotten. Since defensive tests are performed before the instruction is executed, the proof is trivial, and all the proof obligations can be discharged by the *Atelier B* automatic prover.

9.2. The aggressive interpreter

The refinement corresponding to the replacement of runtime tests by static properties generates most of the proof obligations. Two kinds of proof obligations are generated for each instruction, corresponding to the case where the defensive tests pass and to the case where the defensive tests fails.

For the first case, the remaining proof obligations require proving that the invariant relating the content of the dynamic and static frames to the static ones still holds. That corresponds to prove that the new stack and the new frame are equal to the static stack and the static frame for the new program counter. Such a proof obligation corresponding to the *iload* opcode is given Fig. 22.

Using the fact that the *ILOAD* opcode belongs to the *OP_NEXT* group, and using the fact the *stack_checked* implies *static_stack_checked*. The demonstration is

$$
\begin{array}{l}
opcode(apc) = ILOAD \wedge \\
flow\_checked = \mathbf{TRUE} \wedge \\
stack\_checked = \mathbf{TRUE} \wedge \\
frame\_checked = \mathbf{TRUE} \\
\Rightarrow \\
stack\_type\_s(\text{succ\_pc}(apc)) = \\
\qquad tpushed(opcode(apc)) \wedge (stack\_type \downarrow \mathbf{size}(tpopped(opcode(apc))))
\end{array}
$$

Fig. 22. Example proof obligation for stack invariant preservation.

$$
\begin{array}{ll}
opcode(apc) = ILOAD \wedge & \\
flow\_checked = \mathbf{TRUE} \wedge & (1) \\
stack\_checked = \mathbf{TRUE} \wedge & (2) \\
frame\_checked = \mathbf{TRUE} \wedge & (3) \\
\neg(BYTE\_to\_unsigned(parameter(apc, 1)) \in 0..max\_locals\text{-}1 \wedge & \\
\quad frame\_type(BYTE\_to\_unsigned(parameter(apc, 1))) = & \\
\qquad\qquad frame\_type\_used(opcode(apc)) \wedge & \\
\quad \text{succ\_pc}(apc) \in opcode\_locations \wedge & \\
\quad \text{can\_update\_stack}(apc)) & (4) \\
\Rightarrow & \\
apc = \text{succ\_pc}(apc) &
\end{array}
$$

Fig. 23. Proof obligation for dynamic tests removal.

straightforward (although requiring user interaction). The other proof obligations are similar and require using *flow_checked*, *stack_checked* or *frame_checked* and the corresponding implied properties. Those proof obligations allow verifying the consistency between the behaviour of the defensive machine and the static constraints expressed as invariant.

The second type of proof obligation handle the case where the defensive tests fail. This case cannot happen, and the proof can be performed by finding a contradiction within the hypothesis. An example of such a proof obligation is given in Fig. 23.

In this proof obligation, (4) correspond to the negation of the test performed by the defensive tests. The proof can be discharged by proving that this implies a contradiction with (1), (2), and (3): if the method has been successfully verified, then the predicate (4) is true. This corresponds to prove that no runtime error can occur for this instruction if the program has been successfully checked.

9.3. Proof of the verifier and interpreter inclusion

As for the proof of the static checks, we can distinguish between two distinct kinds of proof obligations: proof obligations that aim to ensure that the preconditions of the called interpreter operations are true, and proof obligations ensuring that the gluing invariant still holds after the operation is performed. Fig. 24 corresponds to a proof obligation enforcing the precondition of the called operation.

This proof obligation can be demonstrated using the hypothesis (1), (2) or (3), and using the definition of *static_flow_checked*, *static_frame_checked* or *static_stack_*

$$
\begin{array}{ll}
opcode(dpc) = ILOAD \wedge & \\
flow\_checked = \mathbf{TRUE} \wedge & (1) \\
stack\_checked = \mathbf{TRUE} \wedge & (2) \\
frame\_checked = \mathbf{TRUE} & (3) \\
\Rightarrow & \\
\text{succ\_pc}(dpc) \in \mathbf{NAT} &
\end{array}
$$

Fig. 24. Proof obligation enforcing preconditions.

$$
\begin{array}{ll}
opcode(dpc) = ILOAD \wedge & \\
var\_value = frame\_value(BYTE\_to\_unsigned(parameter(dpc, 1))) \wedge & \\
flow\_checked = \mathbf{TRUE} \wedge & (1) \\
stack\_checked = \mathbf{TRUE} \wedge & (2) \\
frame\_checked = \mathbf{TRUE} & (3) \\
\text{succ\_pc}(dpc) \in \mathbf{NAT} & \\
var\_value \rightarrow stack\_value \in \mathbf{seq}(\mathbf{INT}) \wedge & \\
\Rightarrow & \\
\mathbf{size}(tpushed(opcode(apc)) & \\
\hat{}(stack\_type \downarrow \mathbf{size}(tpopped(opcode(apc))))) = \mathbf{size}(var\_value \rightarrow stack\_value) &
\end{array}
$$

Fig. 25. Proof obligation enforcing gluing invariant.

$$
\mathbf{size}([integer]) + (\mathbf{size}(stack\_value)\text{-}\mathbf{size}([])) = 1 + \mathbf{size}(stack\_value)
$$

Fig. 26. Rewritten goal.

checked. In this case, from the definition of *static_flow_checked*, the hypothesis (1), and since *apc* is equal to *dpc*, it is clear that succ_pc(*dpc*) is included in *opcode_locations*. The demonstration can then be carried out by showing that *opcode_locations* is a subset of the natural numbers.

A sample proof obligation ensuring the invariant conservation for the *iload* instruction is given in Fig. 25. The proof requires using the gluing invariant **size**(*stack_type*) = **size**(*stack_value*), and the images of *tpushed* and *tpopped* for the *iload* opcode. By replacing *tpopped*(*opcode*(*apc*)) and *tpushed*(*opcode*(*apc*)) by their values, and using the previous gluing invariant, the goal can be rewritten as shown in Fig. 26.

From this, it is clear that the proof obligation holds. However, achieving the proof from this point still involves quite a number of proof commands, especially if the demonstration has to be generalised.

9.4. Proof generalisation

For this specification, the main goal is to limit the cost of the proof process. We focus on obtaining similar proof obligations, so that a single demonstration could be used to demonstrate several similar proof obligations. This is achieved by specifying

PO without groups	*PO with groups*
(1.1) $\forall pc.((pc \in \mathbf{dom}(method) \wedge$ $opcode(pc) = ILOAD)$ $\Rightarrow$ $pc + 1 \in opcode\_locations) \wedge$ (1.2) $opcode(apc) = ILOAD \wedge$ (1.3) $apc \in \mathbf{dom}(method)$ $\Rightarrow$ $apc + 1 \in opcode\_locations$	(2.1) $OP\_NEXT = \{\ldots, ILOAD, \ldots\} \wedge$ (2.2) $\forall pc.((pc \in \mathbf{dom}(method) \wedge$ $opcode(pc) \in OP\_NEXT)$ $\Rightarrow$ $\text{succ\_pc}(pc) \in opcode\_locations) \wedge$ (2.3) $opcode(apc) = ILOAD \wedge$ (2.4) $apc \in \mathbf{dom}(method)$ $\Rightarrow$ $\text{succ\_pc}(apc) \in opcode\_locations$

Fig. 27. Comparison between proof obligations with and without using group.

opcodes properties and constraints in a generic way. This involves grouping opcodes by properties, but also using generic description. For example, using the functions *tpushed* and *tpopped* allows specifying nearly all operations that manipulate the stack in the same way.

To illustrate the advantages of using a generic specification, Fig. 27 depicts two simplified proof obligations: the first one corresponds to a specification that does not group opcodes, and the second one to the specification previously described.

Discharging the proof obligation without grouping is quite straightforward: it involves using hypothesis (1.2) and (1.3) with hypothesis (1.1). However, in the Java Card case, there will be one hypothesis similar to (1.1) by opcode, and the automatic prover will not be able to choose the right one, requiring user interaction. Moreover, as the opcode considered is explicitly used, this interaction will be required for every opcodes. For the complete Java Card interpreter, this means that proving this property for each opcode will need approximately 200 different, but very similar proofs with user interaction.

In the case where groups are used, the hypothesis $opcode(apc) \in OP\_NEXT$ (2.5) can be added. This hypothesis is automatically accepted by the prover thanks to (2.1) and (2.3), and user interaction will not be needed. The new hypothesis (2.5) can then be used with hypothesis (2.2) to discharge the goal. The important point is that the commands used to demonstrate this goal does not consider the opcode names, and can be directly reused to prove similar proof obligations for opcodes that are elements of the set OP_NEXT. This means that there will be one user interaction for nearly two hundred proof obligations. This is all the more important, since the response times of the interactive prover can be very large for such a specification. Another important point with opcode groups is that it also reduces the number of predicates within the invariant. This reduction drastically increases the performance of the tool.

9.5. Impact of the tool used

Although the application of B on such a large system was successful, there are some points that have been encountered with the tool used and that could be improved for using B more efficiently with large or complex specifications.

Especially, it seems that a more elaborated proof language would greatly simplify handling such cases where lots of similar proofs have to be performed. For example, the current language has support for applying a sequence of proof commands to each proof obligation, or to each subgoal. However, it has no conditional constructs, so the different proof obligations have to be proved with exactly the same sequence of commands. An extended language would greatly simplify the writing of generic proofs, and would allow handling complicated proofs without resorting to add new rules or theories to the prover.

Another point with the prover used is that it is very well automated, and can easily discharge simple proof obligations. This is a useful feature for handling large specifications, since it allows discharging lots of trivial proof obligations. However, it appears that this advantage can turn into a disadvantage, since the automated prover can spend large amount of time on proof obligations that are too complicated to prove. Thus, paradoxically, other tools that are less automated, but provide more control on the proof process could be more appropriate for handling such specifications.

The prover also impacted how the specification has been formulated. For example, some invariants have been rewritten to better suit the normalisation used by the prover. This improved the efficiency of the proof process by simplifying the proofs. However the specification was sometime less readable, and in some cases this lead to tradeoffs between the readability of the specification and its ease of proof. So, writing specifications that are too much geared towards the prover used may lead to maintainability problems. For small specifications we found better to keep highly readable specifications, even at the expense of a more expensive proof phase. For larger specifications, although compromises can be made, it seems better to extend the prover with additional rules proved manually.

A similar trick that has fewer impacts on readability is introduced by redundant invariants. Those invariants correspond to properties that have explicitly been added as invariant, although they could have been deduced from the existing invariants. They can be used to improve the automatic prover's efficiency, or to simplify the interactive proof, by providing additional hypothesis. Such invariants have been added in the specification when this was more efficient than demonstrating them.

10. Conclusion

This study shows that it is possible to model large systems with B. The main solution used to handle scalability issues was to group opcodes by properties in order to obtain generic proofs, while retaining an individual description of opcodes encompassing their implementation details.

However, those gains have to be balanced by the fact that the proof obligations are often more complicated to prove, and the initial proof can take some time to be carried out. Moreover, all the Java Card opcodes cannot fit in a group, and some opcodes will still need to be treated as special cases. However, using groups allows us to treat all similar opcodes in a unified manner, and to focus on opcodes that need special treatment. Such unified handling will be especially useful when adding special case

opcodes, such as those dedicated to subroutine handling. It will allow treating those instructions peculiarities specifically, while treating the other ones in a generic manner.

Another point is that it appears that machine checked proofs are more difficult to achieve than hand written proofs: a proof step in an hand written proof can correspond to several proof steps in a machine checked proof. However, although they are more tedious to achieve, machine checked proofs still remain safer than hand written proofs: proving the specification using the *Atelier B* tool allowed us to find a few minor errors in proofs previously done by hand. Although the erroneous proofs were not formally written proofs, but proofs mostly used as preliminary sketch of the machine checked proof, this emphasized the usefulness of performing machine checked verification.

Proving the correctness and the soundness of the type system is a first step to a certification of Java Card. Other parts of the security policy are implemented by different functions such as the firewall, that controls access policies. As one of the common criteria requirements is to guarantee the coherence of all the security mechanisms, it is needed to integrate this model into a more generic model encompassing the whole security policies.

Future works will focus on integrating the firewall specification defined in [13] with the interpreter. Then, the model will be extended in order to model the complete Java Card interpreter. This will allow, not only proving the soundness of the byte code verifier and of the interpreter, but also will ensure the correctness of their implementation.

Acknowledgements

Thanks to G. Mornet and L. Casset for their work on the model, discussions and feedback.

References

[1] M. Abadi, R. Stata, A type system for byte code subroutines, in: Proc. 25th ACM Symp. on Principle of Programming Languages, January 1998.

[2] J.R. Abrial, The B Book, Assigning Programs to Meanings, Cambridge University Press, Cambride, 1996.

[3] M. Alberda, P. Hartel, E. de Jong, Using formal methods to cultivate trust in Smart Card operating system, in: Proc. Second Smart Card Research and Advanced Applications Conference (CARDIS'96), Amsterdam, Netherlands, September 1996, pp. 111–132.

[4] L. Casset, J.-L. Lanet, A formal specification of the Java byte code semantics using the B method, in: Proc. ECOOP'99 Workshop on Formal Techniques for Java Programs, June 1999.

[5] Cohen, Defensive Java Virtual Machine Specification, URL: http://www.cli.com/software/djvm.

[6] E. Denney, T. Jensen, Correctness of Java Card method lookup via logical relations, in: Smolka (Ed.), European Symp. on Programming (ESOP 2000), Lecture Notes in Computer Science, vol. 1782, Springer, Berlin, March 2000, pp. 104–118. URL: http://www.irisa.fr/lande/jensen/papers/esop00.ps.

[7] S.N. Freund, J.C. Mitchell, A type system for object initialization in the Java byte code language, in: Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications, October 1998.

[8] S.N. Freund, J.C. Mitchell, Specification and verification of Java bytecode subroutines and exceptions, Stanford Computer Science Technical Note, August 1999.

[9] A. Goldberg, A Specification of Java Loading and Byte Code Verification, Kestrel Institute, December 1997. URL: http://www.kestrel.edu/HTML/people/goldberg.
[10] Y. Gurevitch, C. Wallace, Specification and verification of the windows card runtime environment using abstract state machines, Microsoft Research Technical Report. http://www.eecs.umich.edu/gasm/papers/wincard.html.
[11] J.L. Lanet, A. Requet, Formal proof of smart card applets correctness, in: Quisquater, Schneier (Eds.), Third Smart Card Research and Advanced Application Conference (CARDIS'98), Louvain-la-Neuve, Belgium, September 1998, Lecture Notes in Computer Science, vol. 1820, Springer, Berlin, pp. 85–97.
[12] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Addison Wesley, Reading, MA, 1996.
[13] S. Motré, Formal Proof of the Applet Firewall, AFADL 2000, Grenoble, France, February 2000. URL: http://www-lsr.imag.fr/afadl/Programme/Articles/.
[14] T. Nipkow, D. Oheimb, Javalight is type-safe-definitely, in: 25th Proc. ACM Symp. on Principle of Programming Languages, January 1998.
[15] J. Posegga, H. Vogt, Byte code verification for Java Smart Cards based on model checking, in: Quisquater, Deswarte, Meadows, Gollmann (Eds.), Proc. 5th European Symp. on Research in Computer Security (ESORICS 98), Louvain-la-neuve, Belgium, September 1998, Lecture Notes in Computer Science, vol. 1485, Springer, Berlin, pp. 175–190.
[16] C. Pusch, Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL, in: Cleaveland (Ed.), TACAS 1999, Amsterdam, The Netherlands, March 1999, Lecture Notes in Computer Science, vol. 1579, Springer, Berlin, pp. 89–103, URL: http://www.in.tum.de/~pusch/.
[17] Z. Qian, Least types for memory locations in Java byte code, Kestrel Institute, Technical Report, 1998.
[18] S. Stepney, D. Cooper, Formal methods for industrial products, in: Bowen, Dunne, Galloway, King (Eds.), ZB'2000, New York, England 2000, Lecture Notes in Computer Science, vol. 1878, Springer, Berlin, pp. 364–393.
[19] Sun Microsystems, Java Card 2.1 Virtual Machine Specification, March 1999.
[20] D. Syme, Proving Java type soundness, Technical Report, University of Cambridge, Computer Laboratory, 1997.