Formal Proof of Smart Card Applets Correctness

Jean-Louis Lanet and Antoine Requet

Gemplus Research Group,
Av du Pic de Bertagne,
13881 Gémenos Cedex France
{jean-louis.lanet,antoine.requet}@gemplus.com

Abstract. The new Gemplus smart card is based on the Java technology, embedding a virtual machine. The security policy uses mechanisms that are based on Java properties. This language provides segregation between applets. But due to the smart card constraints a byte code verifier can not be embedded. Moreover, in order to maximise the number of applets the byte code must be optimised. The security properties must be guaranteed despite of these optimisations. For this purpose, we propose an original manner to prove the equivalence between the interpreter of the JVM and our Java Card interpreter. It is based on the refinement and proof process of the B formal method.

Keywords. Java byte code, security, optimisation, formal specification.

1 Introduction

The use of Java for the next generation of smart cards offers the possibility to download executable code. This possibility increases the flexibility to update the contents of a smart card, but raises a risk of loading a hostile applet. Such an applet could access or modify part of the system. In order to ensure a safe execution, several conditions must be verified on the execution environment. Applet properties must be conscientiously checked.

Security in a smart card has several aspects. As applets often need to exchange information, some mechanism must be set up in order to avoid unauthorised information flow. Those mechanisms can be implemented within hardware devices (MMU) or in software. Java and its virtual machine provide by themself several properties that can ease the implementation of such a mechanisms. For example, the lack of pointer arithmetic associated with a strong check on the typing prevents an applet from forging an address and from scanning the memory space of the smart card bypassing the execution mechanisms. Such properties are enforced by a byte code verifier.

However, due to the size and performance limitation of smart cards, such a verifier cannot be embedded in the card. Alternative ways of ensuring that the executed byte code is valid must be used. A pragmatic approach is to use an off-line verifier, and to digitally sign verified applets. This approach has several advantages, for example, it only requires cryptographic mechanisms to be implemented within the card.

Another security aspect is linked to the modification of the Java virtual machine to better suit the smart card constraints. This optimisation allows to reduce the Java byte code size in order to load more applets into the smart card. So, it is necessary to

J.-J. Quisquater and B. Schneier (Eds.): CARDIS 2000, LNCS 1820, pp. 85–97, 2000.
© Springer-Verlag Berlin Heidelberg 2000

ensure that those optimisations do not weaken the type system, and do not introduce security holes. This is all the more difficult, as it is not possible to use the Java byte code verifier at this point. Those optimisations introduce two new problems :

- ensuring that the planned optimisations do not modify the byte code properties and that the transformed program is equivalent to the original one,
- validating the optimisation process, making sure that the optimisations are correctly applied.

The next paragraph describes the different transformations and optimisation processes. Then we describe the different approaches to formally specify the interpreter. In the fourth paragraph, we express the different properties which must be verified. After a brief introduction to the B Method we present our approach.

2 Definition of the Problem

Two different operations are performed before a Java applet can be loaded into the card : the conversion and the optimisations. The goal of the transformations is to translate the Java byte code into Java Card byte code, while the optimisation process tries to minimise the size of the code and improve its performance. Both the size of the code and the required run time memory are optimised although reducing the run time memory is more important. There are several levels of optimisation.

The first one consists in reducing the size of the frame by adjusting the variable locations and using the overlay technique. This technique tries to assign the same memory location to variable that are not used in the same time. The second level deals with local optimisations like peephole, inlining and constant propagation. If the inlining increases the size of the EEPROM in the other hand it can reduce the RAM utilisation which is of paramount importance. The transformer can be split into several modules :

- the analyser, that performs the data flow analysis in order to derive the static type for each memory location. Several information are obtained by this function :
  - the type of each element of the operand stack at each program step, i.e. the stack types,
  - the type of each local variable at each program step, i.e. the frame types,
- the converter, that translates the Java instructions into Java Card instructions. The translations are always one to one mapping.
- the data optimiser. This module modifies the frame in order to adjust local variables in term of their size and use (overlay),
- the local optimiser. It performs some peephole transformations and the inlining (if necessary) of private methods.

In this paper, we are interested in formally specifying transformations (conversion and data optimisation) and in proving that they preserve the security properties of the byte code. We consider that the data flow analysis has been done and that the static type system can be trusted. The formal specifications of the Analyser and the Local Optimiser will be done in a further work.

Currently, the Java Card byte code is proprietary which allows to modify the JVM implementations to optimise memory accesses by specialising instructions. In the current model, only a few instructions are modified. They deal with the field and local

variable access. The conversion process replaces one instruction by another one without modifying the method call convention.

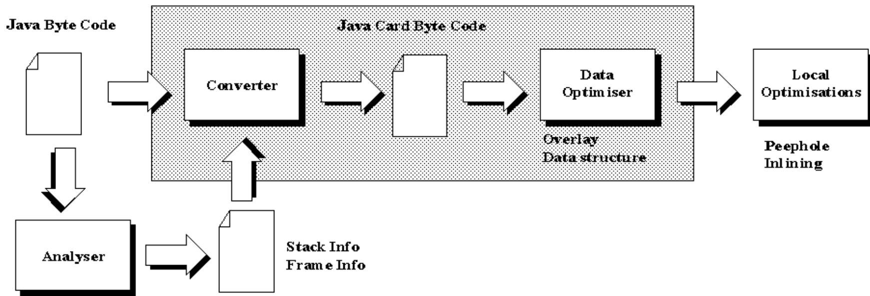

Fig. 1. The Complete Transformation Scheme.

The technique described here is to prove that the byte code interpreter of the smart card is a valid refinement of the Java interpreter. For this purpose we formalise the operational semantics of the instructions and the static constraints of the Java byte code. We express by invariants how to refine it into a Java Card interpreter. We use the B Method to state and prove the different invariants between the two interpreters.

3 Related Work

In our approach we formally specify a part of the Java virtual machine. The semantics of the Java virtual machine has been described by [Qia-97]. In this approach the author considers a subset of the byte code and tries to demonstrate the run time type correctness from the static typing. He provides an operational semantics of the Java byte code and a static inference system. This approach is very close to ours.

Another approach different from ours is described in [Coh-97]. The author gives a formal model of a subset of the Java virtual machine called defensive JVM. The run time checks imply a type safe execution. Our approach is different in the way that we use a static type inference to guarantee run time type errors.

Our approach is partially inspired by the works of [Sta-98] and [Fre-98]. Both of them define a subset of the JVM to prove some properties. Stata et al. define a part of the byte code verifier to focus on the verification of the subroutine whereas Freud et al. verify the correctness of object creation and initialisation.

4 Byte Code Properties

4.1 The Java Card Subset of Java

Due to the physical limitations of smart cards, some features of Java are too expensive to be implemented. Several restrictions have been imposed on the use of Java such as:

- no support for multidimensional arrays and floating point numbers,
- no multithreading,
- no garbage collection mechanism,
- no dynamic loading of class files.

4.2 Verifications Needed

The byte code verifier enforces static constraints on the Java byte code. These constraints rule out type errors (e.g. dereferencing an integer), access control violations (e.g., accessing a private method from outside its class), object initialisation failures (e.g., accessing a newly allocated object before its constructor has been called) and some other dynamic errors. [Yel-96] describes the byte code verifier in an informal manner. Given its importance for security, the current description of the verifier is not sufficient, and we think that an extension to this description using a formal technique of the to-be-checked properties would prove to be useful.

As seen previously, the Java byte code is compiled into a specialised byte code interpreted by the smart card JVM. This byte code is then loaded into the smart card. As the performed optimisations modify the type system of the byte code, several properties must be checked again after the optimisation process. Moreover, after the optimisation process, new checks have to be done in order to verify that the applet respects some specific smart card properties such as correct memory access and restricted loops.

5 The B Method

The B Method is a formal method for software engineering developed by J.R.Abrial [Abr-96]. It is a *model oriented* approach to software construction. This method is based on the set theory and the first order logic. The basic concept is the *abstract machine* which is used to encapsulate the data describing the state of the system. Invariants can be expressed on the state of the machine which can only be accessed by the specified operations.

The abstract machine is refined by adding specification details. Several steps can be used before reaching the implementation level where the specification is detailed enough to generate code. The refinements and the implementation have other invariants which express relations between the states of the different refinements.

The proof process is a mean to check the coherence among the mathematical model, the abstract machine and the refinements. This way, it is possible to prove that the implementation is correct according to its specification. The tool *AtelierB* generates the proof obligations (PO) of the specification according to the mathematical model. A theorem prover is provided to discharge automatically the proof obligations and an interactive theorem prover allows the user to intervene in the proof.

6 The Model

6.1 Model Representation

As explained in [Gol-97] and due to the fact that the optimisations performed do not alter the method call convention, we do not need to model the complete JVM. We only need to model a subset of the JVM interpreter corresponding to a method. We specify our interpreter in terms of a state machine and transition rules. The transitions

define how the execution of an instruction changes the state of the machine under some pre-conditions.

To verify the feasibility of this approach we consider only a subset of the instructions of the JVM. We use the same subset as [Fre-98]. This subset contains simple operations on stack manipulation (*OP_PUSH0*, *OP_POP*), stack operation (*OP_INC*), data transfer (*OP_STORE*, *OP_LOAD*), object manipulation (*OP_NEW*, *OP_INIT*, *OP_USE*), control (*OP_IF*) and return instruction (*OP_HALT*). In the next step we will extend the set to the complete set of the Java Card instructions.

The following machine gives a partial description of the state of our JVM. A method is defined as a pair of sequences. One of them represents the method opcodes and the second the parameters used by the opcodes. Apart from the typing invariants we state that all the domains are equal since all the information have to be available for each method instruction.

```
INVARIANT
  opcodes       ∈ seq (OPCODES_JVM0)   ∧
  parameters    ∈ seq (PARAMETERS)   ∧
  types_stacks ∈ seq (TYPING_STACK))       ∧
  types_frames ∈ seq ( NATURAL ⇸ TYPING_INFO)  ∧
  dom(opcodes)       = dom(parameters) ∧
  dom(types_stacks) = dom(opcodes) ∧
  dom(types_frames) = dom(opcodes) ∧
  size(opcodes) > 0
```

The variables *types_stacks* and *types_frames* are the result of the type inference. They provide typing information about the stack and the frame for each instruction. For verifying our type system we define the type *TYPING_STACK* which is a sequence of primitive types representing the type of elements stored in an operand stack. For each instruction the *type_frame* variable represents a partial map from local variable numbers to types. We give hereafter an example to illustrate the information provided by the type inference. Note that there are several possible *type_frame* variables corresponding to this program.

Table 1. Type Inference Information for a Method.

instructions	*type_stack*	*type_frame*
push 0	[ ]	{ }
store 1	[Byte]	{ }
push 0	[ ]	{1|->Byte}
Inc	[Byte]	{1|->Byte}
store 3	[Integer]	{1|->Byte}
load 3	[ ]	{1|->Byte, 3|->Integer}
load 1	[Integer]	{1|->Byte}
Halt	[Integer, Integer]	{ }

6.2 The Static Constraints

In this machine we define the static constraints of our system. As most of the transformations deal with type definition we have to ensure that the refinement of the JVM is correctly typed. In the following machine we express a part of the static

constraints that must always be verified. We give only a subset of the constraints related to the instructions *OP_PUSH0, OP_IF* and *OP_STORE.*

The first theorem deals with the instruction *OP_PUSH0*. At this method line, if the instruction is *OP_PUSH0* and the program counter is valid, then it must imply that the next instruction is also in the domain of the method. It states that a byte has been added on top of the stack for the next instruction. Such instruction has no parameter.

$$
\begin{aligned}
&\forall pc.(pc \in \mathbf{dom}(opcodes) \wedge \\
&\quad opcodes(pc) = OP_PUSH0 \\
&\quad \Rightarrow \\
&\quad \mathbf{size}(parameters(pc)) = 0 \wedge \\
&\quad pc{+}1 \in \mathbf{dom}(opcodes) \wedge \\
&\quad types_stacks(pc{+}1) = Byte \rightarrow types_stacks(pc) \wedge \\
&\quad types_frames(pc{+}1) \subseteq types_frames(pc))
\end{aligned}
$$

In the previous theorem, the last rule express that the typing information of the local variables at the next instruction can be a subset of the one for the current instruction. In the following example, there is a jump (instruction 7) to the instruction 4 and the contents of the local variable 1 differs according the path used to reach this instruction. Depending on the incoming path, this variable can get different types (Byte or ObjectRef). So, it cannot be defined in *type_frame* (4) although it is included in *type_frame* (3).

Table 2. Example of Typing Information that cannot be determined.

instructions	*type_frame*
1 push 0	{ }
2 store 1	{ }
3 push 0	{1|-> Byte}
4 new Object	**{ }**
5 init Object	{ }
6 store 1	{ }
7 if 4	{1|-> ObjectRef}

The instruction OP_IF requires several theorems. Those linked with object initialisation are not presented here. The following theorem checks the correctness of the frame and stack typing and that the target of the jump (*pc*+*parameters*(*pc*)(1)) is included in the method.

$$
\begin{aligned}
&\forall pc.(pc \in \mathbf{dom}(opcodes) \wedge \\
&\quad opcodes(pc) = OP_IF \\
&\quad \Rightarrow \\
&\quad \mathbf{size}(parameters(pc)) = 1 \wedge \\
&\quad \mathbf{head}(types_stacks(pc)) \in INTEGERS_T \wedge \\
&\quad pc{+}1 \in \mathbf{dom}(opcodes) \wedge \\
&\quad types_stacks(pc{+}1) = types_stacks(pc) \downarrow 1 \wedge \\
&\quad types_frames(pc{+}1) \subseteq types_frames(pc) \wedge \\
&\quad pc{+}parameters(pc)(1) \in \mathbf{dom}(opcodes) \wedge \\
&\quad types_stacks(pc{+}parameters(pc)(1)) = types_stacks(pc) \downarrow 1 \wedge \\
&\quad types_frames(pc{+}parameters(pc)(1)) \subseteq types_frames(pc))
\end{aligned}
$$

The typing rule for the *OP_STORE* instruction ensures that the top element of the operand stack belongs to a correct type and the same type is stored in the local variable indexed by *parameters*(*pc*)(1). It also makes sure that this index is an index in the *types_frames* domain.

The B following operator ($\Leftarrow$+) allows us to overload the partial map *types_frames(pc)* by the couple {*parameters*(*pc*)(1)↦**head**(*types_stacks*(*pc*)). The last rule express the possibility that the type may not be determined, as explained previously.

$$
\begin{array}{l}
\forall pc.(pc \in \mathbf{dom}(opcodes) \wedge \\
\quad opcodes(pc) = OP_STORE \\
\quad \Rightarrow \\
\quad \mathbf{size}(parameters(pc)) = 1 \wedge \\
\quad pc+1 \in \mathbf{dom}(opcodes) \wedge \\
\quad \mathbf{size}\ (types_stacks(pc)) > 0 \wedge \\
\quad types_stacks(pc+1) = types_stacks(pc) \downarrow 1 \wedge \\
\quad parameters(pc)(1) \in \mathbf{dom}(types_frames(pc+1)) \wedge \\
\quad types_frames(pc+1) \subseteq (types_frames(pc) \Leftarrow\!\!+ \{\ parameters(pc)(1) \mapsto \\
\quad \mathbf{head}(types_stacks(pc))\ \}))
\end{array}
$$

6.3 The Operational Semantics

We define the state machine of our interpreter by a tuple (*current_instruction, frame, stack*). Each instruction is modelled by an operation that transforms the machine state. Hereafter we show a part of the refinement machine of our abstract machine.

```
REFINEMENT
  jvm_ref1
REFINES
  jvm_base
VARIABLES
   opcodes, parameters, types_stacks, types_frames, max_stack, stack, frame,
 current_instruction
INVARIANT
  stack ∈ seq(INTEGER ) ∧
  frame ∈ NATURAL ⇸ INTEGER ∧
  current_instruction ∈ dom(opcodes) ∧
  size(stack) = size(types_stacks(current_instruction)) ∧
  (∀pc.(pc∈dom(opcodes)
      ⇒(dom(types_frames(pc)) ⊆ dom(frame))))
```

We use variables, definitions and invariants to specify the state :

- *current_instruction* indicates the method line to be executed, the invariant states that *current_instruction* ∈ **dom**(*opcodes*),
- *frame*, is a partial map from the frame position to their values, *frame* ∈ *NATURAL* ⇸ *INTEGER*,
- *stack* represents the operand stack, modelled by a sequence: *stack* ∈ **seq** (*INTEGER*).

The operational semantic of the interpreter is described by B operations. We give hereafter some operations corresponding to the previously described opcodes.

```
OPERATIONS

op_push0 =
  SELECT
     opcodes(current_instruction)=OP_PUSH0
  THEN
    current_instruction := current_instruction + 1 ||
    stack := 0 → stack
  END;

op_if =
  SELECT
     opcodes(current_instruction)=OP_IF
  THEN
     current_instruction :∈ {(current_instruction + 1),
       current_instruction+parameters(current_instruction)(1)} ||
     stack := stack ↓ 1
  END;

op_store =
  SELECT
     opcodes(current_instruction)=OP_STORE
  THEN
     frame:= frame ⩤ {(parameters(current_instruction)(1)↦head(stack))} ||
     current_instruction := current_instruction + 1 ||
     stack := stack ↓ 1
  END;
```

6.4 Specification of the Optimisation

Within the Java VM, all local variables and parameters are stored using four bytes in the frame. This means, for example, that three bytes of memory are lost for a byte variable. This can increase performance on a 32 bit machine where memory consumption is not a problem, but on a smart card the memory is the most valuable resource to spare. The first performed optimisation consists in using the smallest number of bytes to store a local variable. Moreover the references on a smart card are two bytes wide, which can lead to the spare of a couple of byte.

In order to allow such optimisation, the instruction set of the virtual machine is specialised. In our example, the instructions *store* and *load* are replaced by the instructions *store_byte*, *store_short, store_integer*, *store_ref*, *load_byte, load_short, load_integer* and *load_ref*. Those instructions perform the same operations, excepted that they allow to manipulate one, two or four bytes variables, and address the frame in byte units instead of variable numbers.

The second optimisation that can be performed is the *overlay* optimisation. It consists in allocating the same location in memory for two different variables when they are used at different point in the program. This can also be used to allocate a variable at different positions for different instructions. Such optimisations that can be performed in the table 3:

Table 3. Example of Optimisations

Instructions	*Unoptimised frame*	*Optimised frame*	*optimised frame (overlay)*
1 push 0	{ }	{ }	{ }
2 store 1	{ }	{ }	{ }
3 nop	{ 4 |-> Byte }	{ 1 |-> Byte}	{ 1 |-> Byte}
4 load 1	{ 4 |-> Byte }	{ 1 |-> Byte}	{ 1 |-> Byte}
5 inc	{ }	{ }	{ }
6 new Object	{ }	{ }	{ }
7 init Object	{ }	{ }	{ }
8 store 2	{ }	{ }	{ }
...	{ 8 |-> ObjectRef }	{ 3 |-> ObjectRef }	{ 2 |-> ObjectRef }

Applying this optimisation to the frame can reduce the memory needed for the applet. But it can introduce errors leading to type confusion. The use of formal methods can help to ensure that those optimisations do not introduce errors or security holes. In the next refinement we specify what is allowed for reallocating the variables.

We represent the new location of the variable as a partial injection, mapping the number of the variables used in the previous example to their real position in the frame : *optimised_locations*. This variable uses a sequence since it must be defined for each instruction of the method.

$$\begin{aligned}&optimised_locations \in \mathbf{seq}(\ NATURAL \rightarrowtail\!\!\!\!\!\!+\ NATURAL) \wedge\\&\mathbf{size}(optimised_locations) = \mathbf{size}(opcodes) \wedge\\&\mathbf{dom}(optimised_locations(pc)) = \mathbf{dom}(types_frames(pc)))\end{aligned}$$

The invariant ensures that no local variables overlap each other. The following invariant checks that if the type of the local variable is an integer all the four bytes used cannot be allocated to another variable.

$$\begin{aligned}&\forall(pc,ii).(pc \in \mathbf{dom}(opcodes) \wedge\\&ii \in \mathbf{dom}(types_frames(pc)) \wedge\\&Type(types_frames(pc)(ii)) = Integer\\&\Rightarrow\\&optimised_locations(pc)(ii) \geq PrivateDataSize + 4 \wedge\\&optimised_locations(pc)(ii) - 1 \notin \mathbf{ran}(optimised_locations(pc)) \wedge\\&optimised_locations(pc)(ii) - 2 \notin \mathbf{ran}(optimised_locations(pc)) \wedge\\&optimised_locations(pc)(ii) - 3 \notin \mathbf{ran}(optimised_locations(pc)))\end{aligned}$$

The following invariant ensures that the variables are allocated at the same position for each instruction where they can be used.

$$\begin{aligned}&\forall(pc,ii).(pc \in \mathbf{dom}(opcodes) \wedge\\&pc + 1 \in \mathbf{dom}(opcodes) \wedge\\&ii \in \mathbf{dom}(types_frames(pc)) \wedge\\&ii \in \mathbf{dom}(types_frames(pc+1)) \wedge\\&types_frames(pc)(ii) = types_frames(pc+1)(ii)\\&\Rightarrow\\&optimised_locations(pc)(ii) = optimised_locations(pc+1)(ii))\end{aligned}$$

Another allocation constraint is generated by the if opcode : a variable accessible after the branch must be allocated at the same place as it was previously allocated.

$$
\begin{array}{l}
\forall(pc,ii).(pc \in \mathbf{dom}(opcodes) \wedge \\
opcodes(pc) = OP_IF \wedge \\
ii \in \mathbf{dom}(types_frames(pc)) \wedge \\
ii \in \mathbf{dom}(types_frames(pc+parameters(pc)(1))) \\
\Rightarrow \\
optimised_locations(pc+parameters(pc)(1))(ii) = optimised_locations(pc)(ii))
\end{array}
$$

Although those rules are valid for the considered instruction set, the complete Java byte code requires more care, since some instructions can access directly local variables. Moreover, the exception handling and subroutines calls introduce new constraints.

6.5 The Refinement into the Java Card Interpreter

Until now the refinements were used in order to define the interpreter of the JVM by adding successively new details to enrich the specification. The next refinement transforms the JVM specification into the Java Card specification. We use the notion of gluing invariants to specify the conversion. The gluing invariant expresses how to match the variables of the current refinement with the variable of the previous level. The matching is implicit if the variables have the same name or explicit by expressing the relation between them with an invariant.

We define new variables for the method, the frame. We have to define the relation between the old and the new definition of those variables using the notion of gluing invariants. The next invariant specifies the conversion of the if instruction. It is translated without any change like most of the considered instructions.

$$
\begin{array}{l}
\forall pc.(pc \in \mathbf{dom}(opcodes) \wedge \\
opcodes(pc)=OP_IF \\
\Rightarrow \\
pc \in \mathbf{dom}(opcodes_ref2) \wedge \\
opcodes_ref2(pc)=REF2_IF \wedge \\
parameters_ref2(pc)=parameters(pc))
\end{array}
$$

We also define the new set of opcodes of our Java Card interpreter. Some byte code have been added like those manipulating the local variables (e.g. *REF2_STORE_SHORT).*

The instructions manipulating the frame need a specific invariant. The opcode must be translated into its specialised opcode according to the type contained in the frame. The opcode operand ($parameters_ref2(pc)$) has to be adjusted to the new position of the variable in the frame.

$\forall(pc,yy,zz).(pc \in \mathbf{dom}(opcodes) \wedge$
$opcodes(pc)=OP_LOAD \wedge$
$yy=opcodes_ref2(pc) \wedge$
$zz= types_frames(pc)(parameters(pc)(1))$
$\Rightarrow$
$pc \in \mathbf{dom}(opcodes_ref2) \wedge$
$((zz=Integer) \Leftrightarrow (yy=REF2_LOAD_INTEGER)) \wedge$
$((zz=Short) \Leftrightarrow (yy=REF2_LOAD_SHORT)) \wedge$
$((zz=Byte) \Leftrightarrow (yy=REF2_LOAD_BYTE)) \wedge$
$((zz \in \{ObjectRef, UninitialisedObjectRef\}) \Leftrightarrow (yy=REF2_LOAD_REF)) \wedge$
$parameters_ref2(pc)= \{ (1 \mapsto optimised_locations(pc)(parameters(pc)(1))))$

The next invariant expresses the relation between the frame and the new optimised frame (*frame_ref2)*. It states that every known variable has the same value in the new frame as in the old one. In order to reduce the complexity of the model, we choose to model the access to the frame variables by using only one location in the frame. Consequently, the following invariant need only one access to the frame to store any kind of variables.

$\forall ii.(ii \in \mathbf{dom}(frame) \wedge ii \in \mathbf{dom}(types_frames(current_instruction))$
$\Rightarrow$
$frame_ref2(optimised_locations(current_instruction)(ii)) = frame(ii))$

Now we can express the operational semantics of our Java Card interpreter which must be a valid refinement of the Java interpreter despite the conversion. The operation corresponding to the store opcodes has the same behaviour whatever the store opcode considered due to the previous hypothesis.

op_store =
SELECT
$opcodes_ref2(current_instruction) \in \{REF2_STORE_BYTE,$
$REF2_STORE_SHORT, REF2_STORE_INTEGER, REF2_STORE_REF\}$
THEN
$frame_ref2 := frame_ref2 \lhd+ \{ (parameters_ref2(current_instruction)(1) \mapsto$
$\mathbf{head}(stack_ref2)) \} \parallel$
$current_instruction := current_instruction + 1 \parallel$
$stack_ref2 := stack_ref2 \downarrow 1$
END;

6.6 The Proof of the Specification

The AtelierB generated 645 proof obligations, including 260 non obvious PO. Obvious PO are automatically proved by the theorem prover. Using the strongest force, the theorem prover discharged about 60% of the non obvious PO. The remaining PO have been proved manually using the interactive prover. We encountered several times the same kind of PO, so in order the make the proof generic we added extra rules.

For example, most of the proofs need to instanciate the static constraints corresponding to the current instruction. This can be done, of course, manually but

adding the following rule allowed us to prove several PO with the same demonstration.

```
THEORY prhh IS
        binhyp(opcodes(current_instruction$1) = x) &
        binhyp(current_instruction$1 : dom(opcodes)) &
        binhyp(!pc.(pc : dom(opcodes) & opcodes(pc) = x => y))
        =>
        [pc:=current_instruction$1] y
END
```

Finally the whole specification has been proved as shown in the following table. However the manual proof still represents a considerable effort even for this reduced model.

COMPONENT	TC	GOP	Obv	nPO	nUn	%Pr	B0C
constants	OK	OK	1	0	0	100	-
jvm_base	OK	OK	27	86	0	100	-
jvm_ref1	OK	OK	134	44	0	100	-
jvm_ref2	OK	OK	477	124	0	100	-
method_base	OK	OK	2	4	0	100	-
opcodes_base	OK	OK	1	0	0	100	-
opcodes_constraints	OK	OK	1	0	0	100	-
opcodes_jcvm	OK	OK	1	0	0	100	-
types_base	OK	OK	1	0	0	100	-
typing_info	OK	OK	0	2	0	100	-
TOTAL	OK	OK	645	260	0	100	-

7. Conclusions

Byte code verification is an important part of the Java platform. We believe that properties verification is a key point process in the security policy needing the use of formal methods to ensure a high confidence in the optimisation process. However, difficulty arises when this verification is adapted to platforms such as smart cards. The traditional approach using the Java byte code verifier is unusable when the VM is modified. In our approach we replace the use of a dedicated byte code verifier with the formal proof of the transformations.

In this paper, we have presented an original technique to express a formal proof of a part of the optimisation process included into the Gemplus RAD. We gave a part of the formal description of the operational semantics and the static constraints of two interpreters of a subset of Java byte code. By specifying rigorously the transformation process through gluing invariants, we proved on a subset of opcode that we do not modify the type system properties.

We expect to extend this approach to the entire set of Java Card instructions in order to prove our converter and data optimiser. The next step will be to use the gluing invariants to specify the converter itself and automatically generate the code of the transformer. As we use a formal model, we can experiment several optimisation algorithms.

Future works include the specification of two other entities : the peephole optimiser and the type inference system. For the peephole optimiser, we will probably use the specification of the Java Card interpreter as a starting point.

References

[Abr-96] J.R. Abrial The B Book. Assigning Programs to Meanings. Cambridge University Press, 1996

[Coh-97] Cohen, Defensive Java Virtual Machine http://www.cli.com/software/djvm

[Fre-98] S. N. Freud, J. C. Mitchell, A type System for Object Initialization In the Java Byte Code Language http://theory.standford.edu/~freunds

[Gol-97] A. Golberg, A Specification of Java Loading and Bytecode Verification Kestrel Institute, Dec-97 http://www.kestrel.edu/HTML/people/goldberg/

[Har-98] P. Hartel, M. Butler, M. Levy, The operational semantics of a Java Secure Processor.

[Qia-97] Qian A formal specification of Java Virtual Machine Instruction. Technical Report (abstract), Universitat Bremen, 1997 http://www.informatik.uni-bremen.de/~qian/abs-fsjvm.html

[Sta-98] R.Stata, M.Abadi, A Type System for Byte Code Subroutines Proc. 25th ACM Symposium on Principles of Programming Language, Jan-98

[Yel-96] Franck Yellin, Tim Lindholm , The Java Virtual Machine Specification, ed. Addison Wesley, 1996