

# Development of an Embedded Verifier for Java Card Byte Code using Formal Methods

Ludovic Casset

Gemplus Research Laboratory  
Av du Pic de Bertagne,  
13881 Gémenos cedex BP 100  
phone:+33(0)4 42 36 40 98  
fax:+33(0)4 42 36 55 55  
ludovic.casset@gemplus.com

**Abstract:** The Java security policy is implemented using security components such as a Java Virtual Machine (JVM), API, verifier, and a loader. It is of prime importance to ensure that these components are implemented in accordance with their specifications. Formal methods can be used to bring the mathematical proof that their implementation corresponds to their specification. In this paper, we introduce the formal development of a complete byte code verifier for Java Card and its on-card integration. In particular, we aim to focus on the model and the proof of the complete type verifier for the Java Card language. The global architecture of the verification process implemented in this real industrial case study is described and the detailed specification of the type verifier is discussed as well as its proof. Moreover, this paper presents a comparison between formal and traditional development, summing up the pros and cons of using formal methods in industry.

**Keywords:** Byte Code Verification, Formal Methods, B Method

## 1. Introduction

Smart cards have an established reputation for securing data in information systems. These cards lock and protect their contents (data or applications). The strong security of smart cards is linked to their design. All functions are built into a single component: CPU, communication, data and applications. This all fits into 25 square millimeters. *Open* smart cards let you download code onto cards after their issuance (postissuance). They acquire new functions over their lifetime as new applications are uploaded. There is, however, no reason to believe that the uploaded code has been developed using a methodology that ensures its innocuousness. One of the main issues when deploying these applications is to guarantee to the customer that these applications will be safe i.e., that their execution will not jeopardise the smart card's integrity or confidentiality. The Java security policy defines acceptable behaviour for

a program and the properties that such a program must respect. For example, it must not be possible to turn an integer into an object reference since Java is a type-safe language.

A key point of this security policy is the byte code verifier. The aim of a byte code verifier is to statically ensure that the control flow and the data flow do not generate errors. Moreover, in order to perform these verifications, one has to ensure the syntactical correctness of a file sent to the verifier. Correct construction of this file is of prime importance to the security of the system. Formal methods provide the mathematical proof that the implementation corresponds to the specification. We have modelled and implemented a byte code verifier on the full Java Card language, excepting *jsr* and *ret* instructions (they concern subroutines). We show that the implementation is compatible with the constrained context of the smart card and lastly we discuss results. Moreover, formal development is compared to traditional development with an assessment of pros and cons of formal methods in an industrial context.

This paper presents the results of one case study of the Matisse<sup>1</sup> project. This project aims to propose methodologies, tools and techniques focusing on the use of formal methods in industrial contexts. The case study described below concerns the formal specifications and implementation of a Java Card byte code verifier. The formal development was integrated on-card, and we focused on the description and the discussion on the model of the type verifier, and this is the most difficult part of the verification process.

The remainder of this paper is organised as follow. Section 2 explains the aims of the B method, section 3 focuses on the principles of byte code verification and the Java Card context. Section 4 emphasises the model for the byte code verifier. Section 5 provides some quantitative indicators about the development and section 6 is the conclusion.

## 2. The B method

The B method is a model-oriented formal method based on first-order logic, set theory and generalised guarded substitutions. It is fully described in [1]. This method encompasses the entire development process, from the specification down to the implementation. Code can be automatically generated from the implementation.

The primary component of a B model is the *abstract machine*. A B specification is made of one or more abstract machines. An abstract machine encapsulates data, properties and operations that apply to that data. In a sense, abstract machines are conceptually similar to modules or packages.

There are three main parts in an abstract machine:

- **The variables** describe the state of the machine. They can be sets or elements of a set. Those sets include natural numbers, integers and user-defined sets. B distinguishes between *abstract variables*, used for specification purpose only, and *concrete variables* which correspond to variables in the generated code.

---

<sup>1</sup> European IST Project MATISSE number *IST-1999-11435*.

- **The invariant** expresses the properties enforced by the machine. It consists of a predicate expressed on the variables, and it must always hold. It is also used to assign a type to each variable,
- **The operations** provide a way to access and modify the variables of the machine. They usually contain a *precondition*, which is a predicate that must be true when the operation is called.

Abstract machines are gradually turned into implementations using the *refinement* mechanism. Refinement lets us add specification details to an abstract machine, while preserving its properties. Informally, refining an abstract machine consists in replacing the machine by another machine that has the same interface and that preserves the correctness of the abstraction. A special invariant, called the *gluing invariant* is used to describe the relationship between the state of the abstract machine and its refinement.

Finally, *implementations* correspond to the last refinement step of an abstract machine. They must be specified in a subset of B called B0, matching classical imperative language, and used to generate code.

An important point with the B method is that every step can and should be proved. Each specification has an associated set of *proof obligations* corresponding to proofs that must be demonstrated in order to ensure the consistency of the specification.

### 3. Byte code verification

The byte code verification aims to enforce static constraints on downloaded byte code. Those constraints ensure that the byte code can be safely executed by the virtual machine, and cannot bypass the higher-level security mechanisms. The byte code verification is informally described in [9]. It consists in a static analysis of the downloaded applet ensuring that the downloaded applet file is a valid file, there is no stack overflow or underflow, the execution flow is confined to valid byte code, each instruction argument is of the correct type and method calls are performed in accordance with their visibility attributes (`public`, `protected`, etc...).

The first point corresponds to the structural verification, and the next points are performed by the type verification. The next subsections describe in detail the properties ensured by the verifications. Even if this paper focuses on the model and the implementation of the type verifier, we describe the entire verification process used in the case study.

#### 3.1. The structural verification

Structural verification consists in ensuring that the downloaded file is valid. Which is to say, it must describe java classes and byte code, and the information contained in the file must be consistent. For example, this verifier makes sure that all structures have the appropriate size and that required components do indeed exist. These tests ensure that the downloaded file cannot be misinterpreted by the verifier or by the virtual machine.

Apart from the purely structural verification of the binary format, tests focusing on the file contents are also carried out. These tests ensure that there are no cycles in the inheritance hierarchy, or that no final methods are overridden.

In the case of Java Card, the structural tests are more complex than those for Java, since the CAP file format used to store Java Card packages was designed for simple installation and minimum linking. Most references to other components are actually given as offsets in the component.

A CAP file consists of several components with specific information from the Java Card package. For instance, the `Method` component contains the byte code of the methods, and the `Class` component information on classes such as references to their super classes or declared methods.

Therefore in the case of Java Card, we distinguish internal structural verifications from external structural verifications. The internal verifications correspond to the verifications that can be performed on a component basis. Example verification consists in making sure that the super classes occur first in the class component.

External verifications are tests ensuring the consistency between components or external packages. For example, one of those tests consists in checking that the methods declared in the `Class` component correspond to existing methods in the `Method` component.

### 3.2. The type verification

This verification is performed on a method basis, and must be done for each method present in the package.

The type checking part ensures that no disallowed type conversions are performed. For example, an integer cannot be converted into an object reference, downcasting can only be performed using the `checkcast` instruction, and arguments provided to methods have to be of compatible types.

As the type of the local variables is not explicitly stored in the byte code, it is needed to retrieve the type of those variables by analyzing the byte code. This part of the verification is the most complex, and is demanding on both time and memory. It requires that the type of each variable and stack element be computed for each instruction and each execution path.

```
static void m(boolean b) {
    if(b) {
        int i = 1;
    } else {
        Object tmp = new Object();
    }
    int j = 2;
}
```

**Fig.1.** A sample Java method

In order to make such verification possible, the verification is quite conservative about which programs will be accepted. Only programs where the type of each element in the stack and local variable is the same whatever path has been taken to

reach an instruction are accepted. This also requires that the size of the stack be the same for each instruction for each path that can reach this instruction.

Figure Fig.1 shows a sample Java method. The corresponding byte code instructions and types inferred by the verifier are given in figure Fig.2. In this example, depending on the value of the Boolean *b*, an integer or an object is pushed onto the stack and then stored in the local variable *v0*. In both cases, a second integer is pushed onto the stack and stored in the local variable *v0*. This lets us study the typing evolution of the stack and more precisely of the local variable *v0* which can contain either an integer or an object. The goal of the type verifier is to ensure that each branch is correct by determining the type of each stack element and each local variable at any point of the program. Fig.2 indicates the typing value of each element, at each point in the program.

### 3.3. Adaptation to embedded devices

Performing full byte code verification requires large amount of computing power and memory. So different systems have been proposed to allow verification to be performed on highly constrained devices such as smart cards. Those systems rely on an external pre-treatment of the applet to verify. As the type verification is the most resource consuming part of the verification, they aim to simplify the verification algorithm.

Two approaches are usually used: Byte code normalisation and proof carrying code (PCC) or similar techniques. The next subsection introduces those techniques. The proof carrying code technique will be discussed more in detail, since this is the approach that has been developed for the type verifier.

#### **Byte code normalisation**

Byte code normalisation is the approach used by Trusted Logic's smart card verifier [7]. It consists in normalising the verified applet so that it is simpler to verify. More exactly, the applet is modified so that:

- Each variable has one and only one type.
- The stack is empty at branch destinations.

This greatly reduces the memory requirements, since the verifier does not have to keep typing information for each instruction, but only for each variable in the verified method. The computing requirements are also reduced, since only a simplified fixed point computation has to be performed. However, as the code is modified, its size and memory requirements can theoretically increase.

#### **Lightweight byte code verification**

Introduced by Necula and Lee [10], the PCC techniques consist in adding a proof of the program safety to the program. This proof can be generated by the code producer, and the code is transmitted along with its safety proof. The code receiver can then verify the proof in order to ensure the program safety. As checking the proof is simpler than generating it, the verification process can be performed by a constrained device.

An adaptation of this technique to Java has been proposed by Rose [16] and is now used by Sun's KVM [18]. In this context, the "proof" represents additional type

information corresponding to the content of local variables and stack element for the branch targets. Fig.2 depicts the contents of the proof for the previous example method. Those typing information correspond to the result of the fixed point computation performed by a full verifier. In this case, the verification process consists in a linear pass that checks the validity of this typing information with respect to the verified code. In our example, we note that the proof only concerns elements that are a jump target (**endif** and **else**). The types of these elements are extracted from the full type computation. Moreover, in this example, it only concerns the local variable v0, the stack and the other local variables are empty for jump target. Compared to byte code normalisation, lightweight verification requires removing the `jsr` and `ret` instructions from the byte code, and needs temporary storage in EEPROM memory for storing the type information. However, lightweight verification performs the verification as a linear pass throughout the code, and leaves the code unmodified.

.method public static m(Z)V .limit stack 2 .limit locals 1 iload_0 ifeq else iconst_1 istore_0 goto endif <b>else:</b> new java/lang/Object dup astore_0 invokespecial java/lang/Object/<init>()V <b>endif:</b> iconst_2 istore_0 return .end method	Infered types		Proof v <sub>0</sub>
	v <sub>0</sub>	Stack	
	int		
	int	Int	
	int		
	int	int	
	int		
			int
	int	Object	
	int	Object	
	int	Object	Object
	Object	Object	
	top		top
	top	int	
	int		

**Fig.2.** A sample Java bytecode method and its associated proof and type information

### 3.4 Formal studies on byte code verification

A lot of formal work has been done on Java byte code verification. Most of those studies focus on the type verification part of the algorithms.

One of the most complete formal models of the Java virtual machine is given by Qian [14]. He considers a large 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 deduced from the specifications of the virtual machine. In a more recent work [5] the authors also propose a correct implementation of almost all aspects of the Java byte code verifier. They view the verification problem as a data flow analysis, and aims to

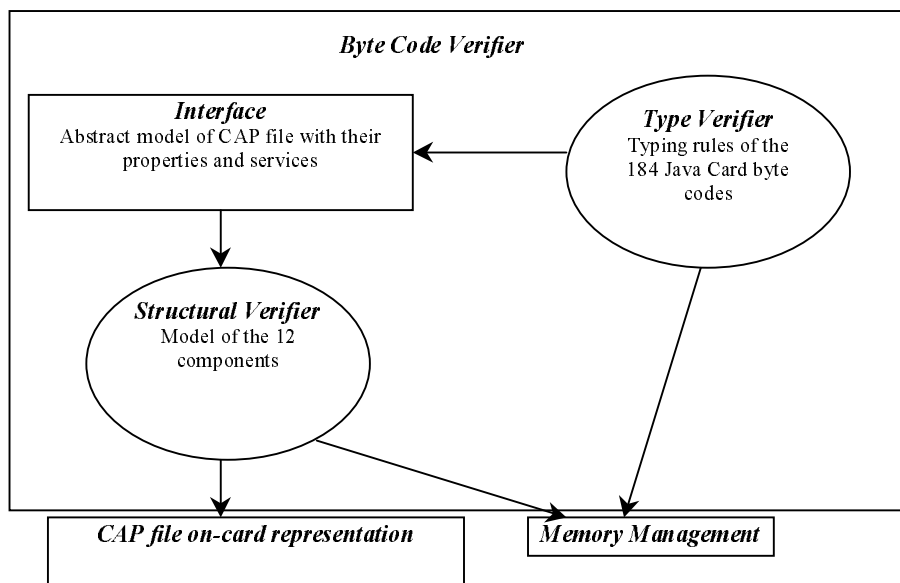
formally describe the specification to extract the corresponding code using the Specware tool.

In the Bali project, Push [12] proves a part of the JVM using the Isabelle/HOL<sup>2</sup> prover. Using Qian's work [14], she gives the verifier specification and then proves its correctness. She also defines a subset of Java,  $\mu$ java [13] and aims to prove properties. More precisely, they formalise the type system and the semantics of this language using the Isabelle theorem prover. In more recent work [11], Nipkow has introduced the formal specification of the Java byte code verifier in Isabelle. The idea is to come up with the generic proof of the algorithm and then to instantiate it with a particular JVM.

Roses' verification scheme has been proven safe using the Isabelle theorem prover by Nipkow [6], and a similar scheme for a Smart Card specific language has been proved correct using B in [15].

Work prior to the one described in this article has also been performed using the B method on the formalisation of a simple verifier [3], and its implementation [4]. Similar work has been done by Bertot [2] using Coq's<sup>3</sup> theorem prover. He proves the correctness of the verification algorithm and generates an implementation using Coq's extraction mechanism.

#### 4. Modeling a type verifier in B



**Fig.3.** The Byte Code Verifier Architecture

<sup>2</sup> Isabelle web site, <http://www.cl.cam.ac.uk/Research/HVG/Isabelle/index.html>

<sup>3</sup> Coq's Proof Assistant web site, <http://coq.inria.fr>

In this section, the modelling of the type verifier is described. We focus on how the architecture was chosen and on the benefits of using formal methods in developing a byte code verifier. Moreover, Fig. 3 indicates that the type verifier is only a part of a complete verifier. In our model, the type verifier relies on services expressed in its interface. It also defines and implements the typing behaviour of the Java Card language, byte-code-by-byte-code. The interface between the structural and the type verifiers constitutes the abstraction of the CAP file. This abstraction is refined in the structural verifier where each component of the CAP file is modelled. Moreover, in the structural verifier, tests indicating that the data received by the smart card represent a CAP file are also implemented.

In this paper, and more precisely in this section, we propose to go deeper into the model of the type verifier and leave aside the structural verifier. The first subsection redefines the type verifier in the general verification scheme. Then, next subsections focus on the type verifier.

#### **4.1 The type verifier: a part of a complete verifier**

As described in section 3, a byte code verifier includes two distinct parts: a structural verifier and a type verifier. These two parts have distinct aspects:

From a purely functional point of view, these parts are the two successive steps of the verification process that can be easily separated.

From an algorithmic point of view, the structural verifier is split into twelve different components that can be modelled separately. Those components correspond to the twelve components defined in the CAP file format. Each component requires a syntactic analysis of the byte stream. On the other hand, the type verifier consists of a linear treatment of a set of byte codes, with a particular treatment for each different byte code according to its respective typing rules.

From a model point of view, as we will develop in this section, models of the two verifiers use the B method quite differently.

We have developed a particular and unique model for the verifier as in our architecture, the type verifier relies on the structural verifier. The structural verifier reveals the properties and the services necessary to the type verifier (Fig. 3).

The type verifier is entirely modelled in B, except that part concerning memory allocations. To access data contained in different CAP file components, an interface is modelled. This interface contains a model necessary for the type verifier. Then, this interface is refined and completed to propose not only the different services dedicated to the type verifier but also the specification of the different tests related to the structural verification. This latter part is not entirely modelled in B. This is due to the fact that as the structural verification contains a syntactical verification of a bytes stream, it is very difficult to propose an abstract representation. Some CAP file components are entirely modelled until an interface is reached that allows us to read a byte into a file. Hence, we show our ability to perform such a model relying on low-level basic blocks. In order to compare the benefits of the modelling, some other components (2 out of 11), are not entirely modelled and are directly implemented in C code. However, all the external tests are modelled in B. These models follow the same

refinement scheme as the one used for the type verifier. In the next subsections, such a scheme is described.

## 4.2 The type verifier model

The aim of the type verifier is to ensure that the typing rules specified by the Java Card language are always respected for each execution of the code. Moreover, it ensures that there is no stack overflow nor underflow during the code runtime. It also ensures the memory confinement. In the specific case of our development, it was decided to include the verification of the *Reference Location* component. In fact, the type verification and the *Reference Location* verification can be simultaneously performed as they both require an analysis of the byte code.

The type verification, according to Sun Specification [17] can be performed method-by-method. For each method, the byte code can be linearly verified thanks to the PCC technique, as described in section 3. Hence, the type verifier complexity increases in linear proportion to the size of the code to verify; although the type verification's search for information within the CAP file can be complex.

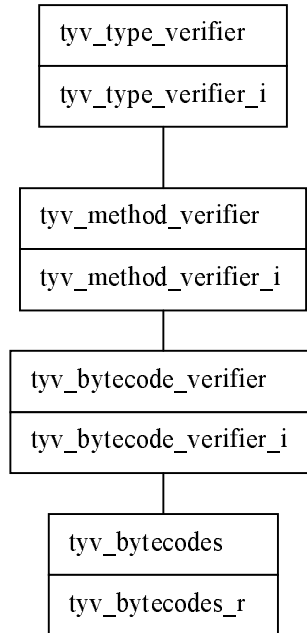
The type verifier is entirely modelled in B. It is composed of an abstract model refined by a concrete model. The abstract model contains the high-level loops, *i.e.* a loop over the methods and then a loop through the byte code of a given method, and the complete specification of all tests that have to be performed on each different byte codes. The concrete model implements the abstract one. In particular it relies on services proposed by the structural verifier and on basic machines that model the volatile memory (RAM) which lets us create temporary variables and tables mandatory to the verification.

### Abstract model

The highest level abstract machine, which defines its interface, remains very simple as it only proposes a single operation returning a Boolean, true if the verification is a success, otherwise it is false. This operation is then implemented by two overlapped loops that call the operation specifying the test related to the byte code instruction being checked. This is summarised by the architecture depicted in **Fig.4**.

The *tyv\_type\_verifier* machine contains a single operation returning a Boolean. This operation is implemented by a loop for all the methods contained in the file to be checked. It is easy to perform since all methods in Java Card are contained in the *Method Component* [17]. The loop calls an operation of the *tyv\_method\_verifier* machine that also returns a Boolean. This latter operation is implemented by a second loop over all the instructions of the method being checked. Finally, the second loop calls an operation of the *tyv\_bytecode\_verifier* machine. This last operation is implemented by a case by case treatment depending the nature of the instruction being checked. In Java Card, there is 184 different byte codes. Hence, there are 184 different cases. In fact, for each different case, a specific operation is called. These operations are basically described in *tyv\_bytecodes* and completely described in its refinement. In this way, the eight abstract machines including four machines, 3 implementations and one refinement contribute to constitute the abstract model of the type verifier. This specification part consists of few properties and the proof process is used mainly to ensure that loops terminate and that the types of the variables are

correct. This model is relatively complex: the *tyv\_bytecodes\_r* refinement by itself is of 5000 lines of B distributed for 90 different operations.



**Fig.4.** Abstract model architecture of the type verifier

### Concrete model

The concrete model implements the above-mentioned abstract model. Therefore, it lets us to obtain the proof ensuring the correctness of the implementation. The implementation relies on services allowing to access information contained in the CAP file. These services are described in an abstract machine used as an interface. This interface contains the variables and the services used by the type verifier. The main interest of this interface is to propose sufficient properties and services to define and implement a type verifier. We can then admit that this interface represents a first draft of what should be a structural verifier. The only part that the type verifier relies on and that is not entirely modelled in B is the dynamic memory allocation. In fact, this part is developed using basic machines: a basic machine consists in defining the specification in B and directly providing its implementation in C. Hence, there is no proof ensuring the correctness of the implementation compared to its formal specification. The reason why this part is not developed in B is that the size of the stack is not static and moreover, it depends on the method being checked. Therefore one needs to propose a dynamic allocation mechanism to build array in memory.

The remainder of the concrete model construction is classical. It is composed of the structural refinement, with the refinement of the SELECT clauses by IF statement. In this first part, loops are introduced to perform compatibility tests for method signatures or for types of the stack. These refinements end on simple update operations on sequences (for the loop) or on partial functions (for local variables). In a

second step, this abstract data is finally refined to obtain the final mapping with a memory pointer.

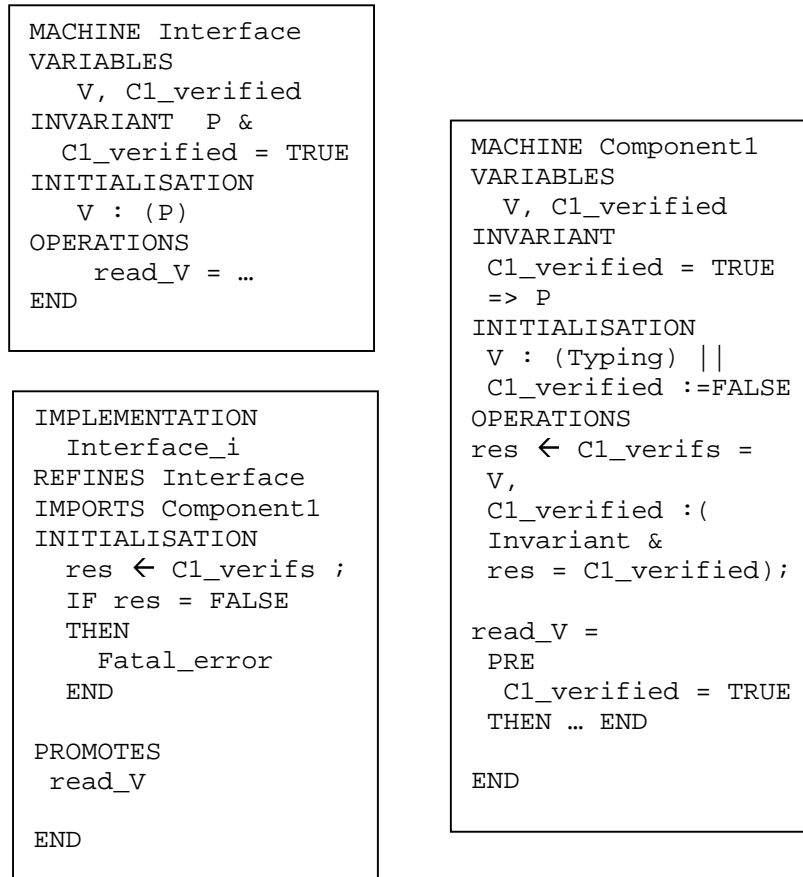
In a partial conclusion, it seems very interesting to use B method in the type verifier context as, from an abstract model we construct a concrete one containing more than 25 machines. Moreover, half of these 25 machines are implementations. The proof of this concrete model ensures us of its validity among the abstract model which is the straight translation of the informal specification.

### 4.3 B architecture of the interface

The interface defined by the type verifier ensures that both verifiers can really work together. The interface, as described in the previous section appears to be the basis for the structural verifier which implements this interface.

The structural verifier architecture is, from our experience, not really conventional, in particular for the use we propose of the `INITIALIZATION` clause. The interface between the type and the structural verifier is in reality an abstract machine containing variables and properties over these variables. These latter properties are used by the type verifier to ensure its correctness. However, these properties are not initially ensured. In fact, the structural verifier needs to perform some tests in order to make sure that all the specified properties are enforced. The internal tests of each component are largely responsible for ensuring these properties. The first solution is to guard each property by the fact that the component ensuring the concerned property has been structurally verified. If the type verifier has as a precondition the fact that all components have been successfully verified, it can now use the properties. However, we have not chosen this solution because there is a major drawback: It complicates the proof by replacing all properties  $P$  by  $(component\_x\_verified = TRUE \Rightarrow P)$ , which is really heavy to handle with the proof process.

The solution (Fig. 5) that we have set up uses the `INITIALIZATION` clause of the interface abstract machine. In this `INITIALIZATION` clause, we call each component internal verification operation. Hence, one ensures that at the initialisation step, in fact after the syntactic verification of all files concerned by the verification, the type verifier, and also the external structural tests, can rely on properties established. In order to allow an abnormal verification termination, in fact if an error is encountered, a special operation is defined, called `Fatal_Error`. When an error is raised by the structural verifier, this operation is called. The consequence is that the verification is stopped and an error code is returned. This ensures that the type verifier is executed only if the structural verifier has successfully accomplished its task. Note that, we develop a prototype. Hence, we need to know why the verifier has stopped. In normal use, a verifier must only return true or false.



**Fig.5.** Interface specification

#### 4.4 Detailed specification

In this subsection, we introduce an example of the detailed specification of a bytecode : `aaload`. We first present its informal specification as it is described by Sun in [17]. We then provide our verifier informal specification, extracted from the previous one, and finally the B model of the operation which checks types when an `aaload` is encountered.

The `aaload` bytecode is used to obtain a reference from an array of reference, the stack contains the reference on the array and the index, representing the location of the reference in the array. After the execution of `aaload`, the stack contains the reference found in the array at the index.

#### 4.4.1 Sun VM specification

We have to extract from the Sun virtual machine specification, the part concerning type verification. Within the Sun specification the type verification and the dynamic behaviour of the VM are melted. In this example (Fig.6), in the *Description* paragraph, the two first sentences describe type verification and the two last sentences describe the dynamic VM execution. The *Stack* paragraph describes the stack evolution. The *Exceptions* paragraph is not really useful, one just has to know that this bytecode can throw exceptions.

<p><b><i>Aaload</i></b> Load reference from array</p> <p><b>Stack</b> ..., <i>arrayref</i>, <i>index</i> . ..., <i>value</i></p> <p><b>Description</b> The <i>arrayref</i> must be of type <b>reference</b> and must refer to an array whose components are of type <b>reference</b>. The <i>index</i> must be of type <b>short</b>. Both <i>arrayref</i> and <i>index</i> are popped from the operand stack. The reference <i>value</i> in the component of the array at <i>index</i> is retrieved and pushed onto the top of the operand stack.</p> <p><b>Runtime Exceptions</b> If <i>arrayref</i> is null, <i>aaload</i> throws a <code>NullPointerException</code>. Otherwise, if <i>index</i> is not within the bounds of the array referenced by <i>arrayref</i>, the <i>aaload</i> instruction throws an <code>ArrayIndexOutOfBoundsException</code></p>
--

Fig.6. *aaload* specification from Sun

#### Informal type verifier specification

Our informal specification (Fig.7) is usually an extraction of the static part of the defensive machine. The evolution of the stack and/or local variables in term of type is given first, then the pre-modification tests describe the test to be performed before modify the stack and/or the local variables. Modification of the stack and/or local variables are then given. The tests that have to be performed after the modification are described in the post modification test.

For this bytecode, two cases are distinguished. The normal case where the reference on the array is not null and the second one when it is null. The type verifier can distinguish between null reference and non null, since null is described with a special type in the lattice. The lattice concerned here is the lattice of the Java Card types as described in [8]. In the Sun specification, the second case is described in the Exception because in this case, at runtime, an exception is produced. We have chosen to model this case as a normal case, where a null reference is pushed onto the stack.

This example shows that we have had to explain some tricky points of the specification and that its has been really useful to write this informal specification before beginning the formalisation, the formalisation is usually a translation of this text.

<p><b><i>aaload</i></b></p> <p>[ ..., refarray class, short ] =&gt; [ ..., ref class ]  [ ..., null, short ] =&gt; [ ..., null ]</p> <p><b>Pre-modification tests:</b></p> <ol style="list-style-type: none"> <li>1. The stack must contain at least two elements</li> <li>2. The two topmost elements of the stack have to be of types compatibles with refarray class and short.</li> </ol> <p><b>Modifications:</b></p> <p>The two topmost elements of the stack are removed. If the second element was a refarray type, then a reference of the same class is pushed onto the stack. Otherwise a type null is pushed.</p> <p><b>Post-modification tests:</b></p> <p>None</p> <p><b>Throws</b></p> <ul style="list-style-type: none"> <li>• NullPointerException</li> <li>• ArrayOutOfBoundsException</li> <li>• SecurityException</li> </ul>
--

Fig.7. Informal specification for aaload verification

**Formal specification**

The next figure (Fig.8) depicts the B operation describing the verifier's behaviour when it is checking an aaload byte code. The different cases are translated with a SELECT clause. This clause allows us to specify action guarded by condition. In this case, all the SELECT are deterministic and the global specification is deterministic.

The operation returns a result, a value is given to the variable containing the result in each branch and the stack is modified in the two correct branches (two elements are popped and one is pushed).

We have three cases, the first two where the byte code is accepted, the last one where it is refused. The first lines of the conditions are exactly the translation of the informal specification, the stack is modelled by a sequence of type, the operator *size*, *last*, *front* are defined in the B language with the usual semantics.

Further explanation is needed for two items within this example:

- The two correct cases contain an additional condition which makes sure that the current pc is in an exception handler. If so, we have to make sure the local variables are compatible with the descriptor associated with the label and also that there are no non-initialised references in the local variables. This must be done for all proof labels of a given handler.
- This test is factored out at the beginning of the informal specifications.
- The second item concerns the class information associated with a reference, as the class information is the same for the *refarray* than for the *ref*, we do not have to pop it just to push it.

```

bb ← verify_aaload =
BEGIN
  SELECT
     $2 \leq \text{size}(\text{stack}) \wedge$ 
     $\text{last}(\text{stack}) = c\_short \wedge$ 
     $\text{last}(\text{front}(\text{stack})) = c\_refarray \wedge$ 
     $(pc \in \text{dom}(\text{exception\_handler}))$ 
     $\Rightarrow$ 
     $\forall \text{label}. (\text{label} \in \text{exception\_handler}(pc)$ 
       $\Rightarrow \text{COMPATIBLE}(\text{loc\_var}, \text{loc\_var\_descriptor}(\text{label}))) \wedge$ 
     $c\_uref \notin \text{ran}(\text{loc\_var})$ 
  THEN
    bb := TRUE ||
    stack := front(front(stack)) ← c_ref
  WHEN
     $2 \leq \text{size}(\text{stack}) \wedge$ 
     $\text{last}(\text{stack}) = c\_short \wedge$ 
     $\text{last}(\text{front}(\text{stack})) = c\_null \wedge$ 
     $(pc \in \text{dom}(\text{exception\_handler}))$ 
     $\Rightarrow$ 
     $\forall \text{label}. (\text{label} \in \text{exception\_handler}(pc)$ 
       $\Rightarrow \text{COMPATIBLE}(\text{loc\_var}, \text{loc\_var\_descriptor}(\text{label}))) \wedge$ 
     $c\_uref \notin \text{ran}(\text{loc\_var})$ 
  THEN
    bb := TRUE ||
    stack := front(front(stack)) ← c_null
  ELSE
    bb := FALSE
  END
END

```

**Fig.8.** Formal specification for aaload verification

## 5. Metrics on the byte code verifier and its development

In this section, we provide metrics about the formal development of the byte code verifier.

Table. 1 synthesises metrics related to the development. In particular, we can note that the structural verifier is bigger than the type verifier. The reason is that the structural verifier contains a lot of tests which require specifications and implementation for each. The type verifier can be seen as a single machine including the typing rules enforced by Java Card. Moreover, the structural verifier contains services on which the type verifier relies. This explains the difference in the number of components as services are organised in different sets.

There are two other results that are remarkable: the first one concerns the number of generated Proof Obligations (POs). The results shows that the type verifier generates

many more POs than the structural verifier. The reason is that there are many more properties in the type verifier than in the structural verifier.

The second results concern the number of C code lines. This number is far smaller than that of corresponding B code. The reason is that in the code translation, only implementations are taken into account. Moreover, INVARIANT clauses within implementations are not translated. This drastically reduces the number of lines translated from B to C.

	Structural Verifier	Type Verifier	Total
<b>Number of lines of B</b>	35000	20000	55000
<b>Number of components</b>	116	34	150
<b>Number of generated POs</b>	11700	18600	30300
<b>POs automatically proved (%)</b>	81 %	72 %	75 %
<b>Project status</b>	90 %	99.9%	95 %
<b>Number of Basic machines</b>	6	0	6
<b>Number of lines of C code</b>	7540	4250	11790
<b>Workload (men months)</b>	8	4	12

**Table. 1.** *Metrics on the formal development of the byte code verifier*

One goal of the Matisse project is to compare formal and traditional developments in order to show the benefits and the drawbacks of using formal techniques in industry. For this reason, two developments were completed concerning only the type verifier: one used formal techniques and the other used traditional techniques. Each development was done by a different person. They both had the same starting point, i.e. an internal document emphasising the requirements of a Java Card type verifier, written in natural language. For each development, we provided a test phase. This step allowed us to check the correspondence between the informal requirements and the code embedded into the smart card. The following tables describe the elements of the comparison.

Table. 3 summarises the number of errors found and the step of the development where they were found. The first conclusion from this comparison is that the formal development produces fewer errors than the traditional development: 56 errors compared to 95. Moreover, only 14 errors were found during the testing step. This is in accordance with the fact that only one week was used to perform the test of the verifier. Compared to the 95 errors of the conventional development and the 3 weeks of testing, there is a significant difference. Unfortunately for the formal development, the proof is very long and costly. However, we believe that this can be decreased thanks to Atelier B improvements and to the development of particular rules and proof tactics. If we manage to capitalise on experience gained in initial developments, we should decrease the time needed by the proof. Moreover, by proposing a methodology adapted to the smart card, the development time required to build models can also be decreased. Using formal methods could then be a real advantage as it is no more costly than conventional development while providing high-quality code.

	<b>Formal development</b>	<b>Conventional development</b>
<b>Number of errors discovered by reading</b>	13	24
<b>Number of errors discovered by proof</b>	29	Not applicable
<b>Number of errors discovered by testing related to the type verifier</b>	14	71
<b>Total Number of errors</b>	56	95

**Table. 2.** *Comparing number of errors for formal and conventional development*

Finally, if the number of errors discovered by reviewing for the formal development is smaller than for the conventional one that is because the modelling activity requires a good understanding of the informal specification and a lot of work required by the refinement method. Errors still exist but the modelling activity helps to clarify the specification by going deeper into the meanings of the specification and thus, reduces the risk of introducing errors. Table. 2 helps us to state that the code produced through a formal process contains fewer errors which indicates a better quality in terms of compliance with the original requirements.

	<b>Formal development</b>	<b>Conventional development</b>
<b>Main development (weeks)</b>	12	12
<b>Proof activity (weeks)</b>	6	Not applicable
<b>Testing (weeks)</b>	1	3
<b>Integration (weeks)</b>	1	2
<b>Number of weeks for the development</b>	20	17

**Table. 3.** *Comparing development time for formal and conventional development*

One of the main results of this case study and of this comparison is that it does not appear unreasonable to use formal methods to develop parts of a smart card operating system (Table. 3). Moreover, even if in this study the time needed for the formal development is greater than that for a conventional development, it is only a difference of three weeks. With a strong involvement in tool improvement and in methodologies, it is possible to be competitive using formal methods. We demonstrate the possibility and the feasibility of developing parts of the operating system or parts of the Java Card Virtual Machine. Finally, we have also shown that we can control the development time of the formal verifier. Thus, with this experiment, it is now possible to be more accurate about the time required for a given development. Finally, Table. 3 allows us to conclude that it is possible to develop a realistic application with an acceptable overhead, i.e., one induced by using formal method is acceptable compared to a conventional development. This conclusion takes into account the fact that the tools and the methodologies are not yet optimised for the smart cards. Hence, we expect to reduce in particular the cost of the proof activity, by developing tools and proof rules to speed up the proof process. Experience gained in the first formal developments should improve our knowledge and speed up future developments.

The last comparison that we can make concerns the efficiency of the produced code, both in terms of the size of the code and in terms of time required to verify an applet. Table. 4 and Table. 5 contain the results of the comparison. The first comment is that the code obtained and translated from the B is acceptable. Concerning the type verifier, we note that the two sizes are similar. This table also shows that the RAM usage is acceptable for a verification algorithm. Note that there is a large range of RAM usage for the conventional verifier as RAM usage is adaptable for this verifier. We also provide the size of the structural verifier but we cannot compare it as it has not been developed yet for the conventional part. The difference that we can note on the total size regarding the other sizes is that in the total size, we include libraries and APIs necessary for both verifiers inside the card. It includes notably the loader, the memory management and the communication.

	<b>Formal development</b>	<b>Conventional development</b>
<b>Type verifier ROM size (kb)</b>	18	16
<b>Structural Verifier ROM size (kb)</b>	24	Not Yet Implemented
<b>Total ROM size (kb)</b>	45	24
<b>RAM usage (bytes)</b>	140	128-756
<b>Applet code overhead (%)</b>	10-20	0

**Table. 4.** *Comparing code size for both developments*

Table. 5 proposes a comparison of execution time for a set of applets. Note that the two implementations, the formal and the conventional ones, are not actually done on the same chip. The formal verifier is implemented on an ATMEL SC 6464 C and the conventional one on an ATMEL SC 3232. The main difference between those two chips is the free memory size (greater in the case of the formal development). Note also that the conventional verifier does not include a structural verifier. Hence, we cannot compare the time for this particular part. However, we provide the information in order to compare the structural verifier's complexity with that of the type verifier.

The main observation about the execution time is that the conventional type verifier is twice as fast as the formal one. There can be several reasons to explain this difference. The first is the difference of memory management between the two developments. The conventional one uses a pointer to access to the memory. In the formal one, the pointer is a translation from the one in B. Therefore, each time we access the memory, there is a translation which costs some time in the execution. Another reason is that the developments were done to optimise the size of the code, not its efficiency. So, when the code is compiled, the compilation directives that are used aimed to optimise the size. The conventional development already takes into account some efficiency optimisations that the formal one does not. Hence the difference of execution time. We have performed some optimisations on the memory management of the formal type verifier. The obtained results show that, with these optimisations, the execution times on the different applets are now similar.

We think that Table. 4 and Table. 5 help us conclude that the code generated with the C code translator from the formal implementation fits the smart card constraints.

	Formal development		Conventional development	
	Type (ms)	Time	Type	Time after optimisation(ms)
<b>Wallet</b>	811	460	318	
<b>Utils</b>	2794	1422	1463	
<b>Pacap</b>	241	110	61	
<b>Interface</b>				
<b>Tic Tac Toe</b>	3555	1372	1102	

**Table. 5.** Comparing verification time for a set of example applets

So, this comparison is encouraging for future formal developments. Of course the data collected concern only a single development. To be more accurate, this kind of comparison should be repeated on other applications. But, this is a realistic application and we think that the comparison is reasonable. Developing methodologies to integrate formal methods into the software development cycle and improving tools to ease and speed up the modelling and the proof activities are possible and may open a new era for software development.

## 6. Conclusion

In this paper, we have presented the formal development of an embedded byte-code verifier for Java Card, and compared it with a similar development using classical techniques.

This experiment shows that applying formal methods for developing industrial applications is possible, even in a constrained context such as smart cards. It also provides an estimation of the development overhead introduced by the use of formal techniques. Moreover, the code generated from the formal models has been translated into C and then integrated into a smart card as the first complete byte code verifier prototype for the Java Card language

Although the workload overhead cannot be neglected, it appears to be small enough to be offset by the increased confidence gained in the development. This is especially important for critical environments where security or safety is a requirement. In fact, we gain confidence in the obtained code, and the byte code verifier prototype can be used as reference implementation for further development.

It also results from this experiment that all the parts of a program do not benefit equally from formal modelling. For example, some low-level modules of the structural verifier were entirely developed with B, requiring significant proof efforts. However, they expose the same kind of bugs as similar parts developed directly in C. So, those modules could have been developed classically without reducing the confidence in the code.

It seems that efficiently applying formal methods will require trade-offs, by identifying which parts of the development require formal development and which parts would incur overhead without providing significant benefits.

## References:

- [1] J.R. Abrial, *The B Book, Assigning Programs to Meanings*, Cambridge University Press, 1996.
- [2] Y. Bertot, *A Coq formalization of a Type Checker for Object Initialization in the Java Virtual Machine*, Research Report, INRIA Sophia Antipolis, 2001.
- [3] L. Casset, J.-L. Lanet, *A Formal Specification of the Java Byte Code Semantics using the B method*, Proceedings of the ECOOP'99 workshop on Formal Techniques for Java Programs, Lisbon, June 1999.
- [4] L. Casset, *Formal Implementation of a Verification Algorithm Using the B Method*, Proceedings of AFADL01, Nancy, France, June 2001
- [5] A. Coglio, Z. Qian and A. Goldberg, *Towards a Provably-correct Implementation of the JVM Bytecode Verifier*, In Proc. DARPA Information Survivability Conference and Exposition (DISCEX'00), Vol. 2, pages 403-410, IEEE Computer Society, 2000.
  
- [6] G. Klein, T. Nipkow, *Verified Lightweight Bytecode Verification*, in ECOOP 2000 Workshop on Formal Techniques for Java Programs, pp. 35-42, Cannes, June 2000.
- [7] X. Leroy, *On-Card Byte Code Verification for Java Card*, Proceedings of e-Smart, Cannes, France, September 2001.
- [8] X. Leroy, *Bytecode Verification on Java smart Cards*, to appear in Software Practice and Experience, 2002.
- [9] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*, Addison Wesley, 1996
- [10] G. Necula, P. Lee, Proof-Carrying Code, in 24<sup>th</sup> ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 106-119, Paris, France, 1997.  
[http://www-nt.cs.berkeley.edu/home/necula/public\\_html/pop197.ps.gz](http://www-nt.cs.berkeley.edu/home/necula/public_html/pop197.ps.gz)
- [11] T. Nipkow, *Verified Byte code Verifiers*, Fakultät für Informatik, Technische Universität München, 2000.  
<http://www.in.tum.de/~nipkow>
- [12] C. Pusch, *Proving the Soundness of a Java Bytecode Verifier in Isabelle/HOL*, In OOPSLA'98 Workshop Formal Underpinnings of Java, 1998.
- [13] C. Pusch, T. Nipkow, D. von Oheimb, *microJava: Embedding a Programming Language in a Theorem Prover*. In Foundations of Secure Computation, IOS Press, 2000.
- [14] Z. Qian, *A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines*. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 271-312. Springer, 1999.
- [15] A. Requet, L. Casset, G. Grimaud, Application of the B Formal Method to the Proof of a Type Verification Algorithm, HASE 2000, Albuquerque, November 2000.
- [16] E. Rose, K. H. Rose, Lightweight Bytecode Verification, in Formal Underpinnings of Java, OOPSLA'98 Workshop, Vancouver, Canada, October. 1998.  
<http://www-dse.doc.ic.ac.uk/~sue/oopsla/rose.f.ps>
- [17] *Java Card 2.1.1 Virtual Machine Specification*, Sun Microsystem, 2000.
- [18] *Connected, Limited Device Configuration*, Specification 1.0a, Java 2 Platform Micro Edition, Sun Microsystems, 2000.