Embedding Formal Development into a Smart Card: The Java Card Byte Code verifier case study

Ludovic Casset
Gemplus Research Lab
BP 100 Gémenos Cedex France

+33(0)4 42 36 40 98
+33(0)4 42 36 55 55
ludovic.casset@gemplus.com

Abstract: The Java security policy is implemented by security components such as the Java Virtual Machine (JVM), the API, the verifier, the loader. It is of prime importance to ensure that the implementation of these components is in accordance with their specifications. Formal methods can be used to bring the mathematical proof that the implementation of these components corresponds to their specification. In this paper, a formal development is performed on the Java Card byte code verifier using the B method. The whole Java Card language is taken into account in order to provide realistic metrics on formal development. The architecture and the tricky points of the development are presented. This formalization leads to an embeddable implementation of the byte code verifier thanks to automatic code translation from formal implementation into C code. We present the formal models, discuss the integration into the card and the results of such an experiment.

Keyword: verification algorithm, B method, byte code verifier, formal methods

1 Introduction

Smart cards have always had the reputation for being secured items of information system. These cards lock and protect their secret (data or applications). The invulnerability of smart cards comes from their conception. Everything is gathered in one single block: memory, CPU, communication, data and applications, everything holds in 25 square millimetres. *Open* smart cards allows to download code on card after its issuance. They have more and more functionality by accepting more than one application. However, there is no reason to believe that the downloaded code was developed following a methodology that guarantees its innocuousness. One of the main issues when deploying these applications is to provide assurance to the customer that the executions of these applications are safe. That is, their execution will not threaten smart card integrity and confidentiality. The Java security policy defines the correct behaviour of a program and the properties that this program must hold. For example, it is not possible to forge an integer into an object reference as 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 check that the flow control and the data flow do not generate an error. Moreover, in order to perform these checks, one has to ensure the syntactical correctness of a file sent to the verifier for verification. Its correct construction is of prime importance to ensure the security of the system. Formal methods are used to obtain 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 which concern subroutines. We finally show that the provided implementation fits the smart card constraints and we discuss results.

This paper presents the results of one case study of the Matisse[1] project. This project aims to propose methodologies, tools and techniques for using formal methods in an industrial concerns. The case study in which we contribute concerns the formal specification and the implementation of the Java Card verifier with the B method [1].
The remainder of this paper is organised as follow. Section 2 focuses on byte code verification principles in the Java Card context. Then section 3 emphases the model of the byte code verifier. Integration of formal development and informal development is discussed in section 4. Section 5 collects some metrics about the development and section 6 concludes.

[1] European IST Project MATISSE number *IST-1999-11435.*

2 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 [11]. It consists in a static analysis of the downloaded applet ensuring that the downloaded applet file is a valid file, there are no stack overflow or underflow, the execution flow is confined to valid byte code, each instruction argument is of the correct type and methods 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 will describe more in details the properties ensured by those verifications.

2.1 The structural verification

The structural verification consists in ensuring that the downloaded file is a valid file. That is, it really describes java classes and byte code, and the information it contains is consistent. For example, it checks that all the structures have the appropriate size and that the required parts exist. Those tests ensure that the downloaded file cannot be misinterpreted by the verifier or the virtual machine.
Apart from the purely structural tests checking the binary format, other tests more related to the content of the file are performed. Those 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 involved than in the Java case as the CAP file format used to store Java Card packages has been designed for simple installation and minimum linking. For example, most references to other component are given as offsets into the component.
A CAP file consists of several components that contains 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 method.
Therefore in the Java Card case, 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 checking that the super classes occur first in the class component.
The external verifications correspond to tests ensuring the consistency between components or external packages. For example, on of those tests consists in checking that the methods declared in the `Class` component correspond to existing methods in the `Method` component.

2.2 The type verification

This verification is performed on a method basis, and has to 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 analysing the byte code. This part of the verification is the most complicated one, and is both time and memory expensive. It requires computing the type of each variable and stack element for each instruction and each execution path.
In order to make such verification possible the verification is quite conservative on the programs that are 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 is the same for each instruction for each path that can reach this instruction.

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

Figure.1. *A sample Java method*

Figure .1 shows a sample Java method. The corresponding byte code instructions and types inferred by the verifier are given on Figure .2.

2.3 *Adaptation to embedded devices*

Performing the 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 details, since this is the approach that has been developed.

2.3.1 Byte code normalisation

Byte code normalisation is the approach used by trusted-logic's [22] smart card verifier [10]. 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 fixpoint computation has to be performed. However, as the code is modified, its size and memory requirements can theoretically increase.

2.3.2 Lightweight byte code verification

Introduced by Necula and Lee [12], the proof-carrying code (PCC) technique consists 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 [19] and is now used by Sun's KVM [21]. In this context, the "proof" consists in additional type information corresponding to the content of local variables and stack element for the branch targets. Figure .2 depicts the content of the proof for the previous example method. Those typing information correspond to the result of the fixpoint 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.

.method public static m(Z)V	Infered types			Proof	
.limit stack 2	v_0	Stack		v_0	stack
.limit locals 1					
iload_0	int				
ifeq else	int	Int			
iconst_1	int				
istore_0	int	int			
goto endif	int				
else:					
new java/lang/Object	int			int	
dup	int	Object			
astore_0	int	Object	Object		
invokespecial java/lang/Object/<init>()V	Object	Object			
endif:					
iconst_2	top			top	
istore_0	top	int			
return	int				
.end method					

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

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.

2.4 *Formal studies on byte code verification*

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 [16]. 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 deducted from the virtual machine specification. In a more recent work [17] 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 [14] proves a part of the JVM using the prover Isabelle/HOL**[8]**. Using Qian works [16], she gives the verifier specification and then proves its correctness. She also defines a subset of Java, μjava [15] and aim to prove properties over it. More precisely, they formalise the type system and the semantics of this language using the Isabelle theorem prover. In a more recent work [13], Nipkow introduces the formal specification of the Java byte code verifier in Isabelle. Its idea is to come 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 [9], and a similar scheme for a Smart Card specific language has been proved correct using B in [18].
Works prior to the one described in this article have also been performed using the B method on the formalisation of a simple verifier [4], and its implementation [5]. A similar work has been performed by Bertot [3] using the Coq **[7]** theorem prover. He proves the correctness of the verification algorithm and generates an implementation using the Coq extraction mechanism.

3 Modelling a byte code verifier in B

In this section, the modelling of the byte code verifier is described. One focuses on how the architecture was chosen and what are the benefits of using formal methods in developing a byte code verifier.
The model of the byte code verifier is developed in two parts: The first one concerns the type verifier and the second one the structural verifier. The type verifier is entirely modelled in B. Its development is made simpler as it relies on many services provided by the structural verifier and expressed through an interface (figure 3). This latter verifier deals more with data representation and low level services provided by the card.. The structural verifier is not entirely developed in B. In fact it relies on basic blocks such as the memory management, and file representation within the smart card. We describe in the following subsections the architecture of the case study, summarised by the following figure (figure 3).

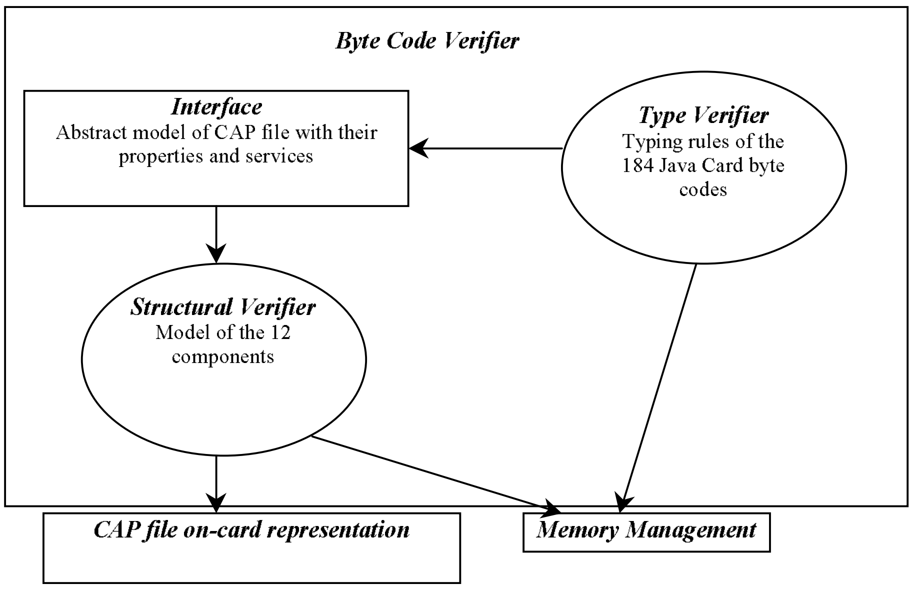
Byte Code Verifier
Interface
Abstract model of CAP file with their properties and services
Type Verifier
Typing rules of the 184 Java Card byte codes
Structural Verifier
Model of the 12 components
CAP file on-card representation
Memory Management

Figure.3. *The Byte Code Verifier Architecture*

3.1 The Type Verifier Model

The type verifier aims to enforce the Java Card typing rules. In fact, it ensures that there is no stack overflow or underflow, an integer cannot be forged into a pointer and so on. The type verifier designed in this section has one more feature: It checks the *Reference Location* component. This verification should be performed in the structural verifier, but as it deals with data manipulated by the type verifier, *i.e.* the byte codes, it is easier to perform it at this stage. The Java Card type verification can be performed method per method. Using the Proof-

Carrying Code technique, the complexity of the verification remains linear with the size of the code. Even if the main loop complexity is linear, there is still high complexity when accessing to data scattered in several components.
The type verifier is entirely modelled in B, from its specification to its implementation. One proves that its implementation is consistent with its specification. The B method allows us to provide a very abstract specification that is split in several modules. In fact, we do not use a simple scheme where the specification is in the abstract machine which is refined and implemented. We provide a formal specification which is made of several modules (abstract machines, refinements and implementation). This formal specification is then refined in order to obtain an implementation.

3.1.1 The type verifier specification

The formal specification, at a very high level is very simple. It states that the verifier must return true or false. Using the refinement process, one clarifies what means returning true and what means returning false. Therefore, the specification of the type verifier is not only the abstract machine but a set of abstract machines, refinements and implementations that describes what the type verifier does. The formal specification is based on several loops for the type verification. The first loop iterates on methods contained into the CAP file being verified. Then a second loop is designed iterating on the different byte codes of the method. One only states that if a method is correct then all its byte codes are correct. Therefore, the specification remains simple. When aiming to ensure correctness of the byte code, a description of each of 184 byte codes is mandatory. This description remains abstract, specifying what does the concerned byte code and what it modifies (the stack, the local variable, etc …). Then, it appears that the formal specification is not contained in a simple abstract machine but distributed in several machines, refinements and implementations.

3.1.2 The type verifier implementation

The formal implementation relies on properties and services defined in the formal specification. It allows to construct the final implementation and also to perform the proof. Finally, it helps ensuring that what is implemented is what has been specified, thanks to the proof. Once the proof is complete, one can have the mathematical proof that the implementation corresponds to the specification. The implementation is expressed in B0, a subset of the B language. It can then be translated into C code as explained in a next section.
When constructing the type verifier model, several services were necessary. First it is necessary to access data contained into *Class*, *Method* and *Descriptor* components [20]. As we are using the Proof Carrying Code technique, it is also necessary to access data of an additional component, i.e. the *Proof component*, that contains pre-computed typing information. Finally, the type verifier uses memory and has to rely on a model of memory management. These services do not need to be related directly to the type verifier. In fact, they correspond to low-level data access or to the CAP file description. An interface has been constructed in order to put together all the requirements and the properties on which the type verifier relies. This interface serves as a basis to construct the structural verifier described in the next subsection.

3.2 The structural verifier Model

This subsection describes the structural verifier. The structural verifier implements the interface of services produced by the type verifier. It also includes internal and external tests (figure 3). Internal tests correspond to tests related to each component of a CAP file, i.e. Applet, Class, etc… We refer to tests between components as external tests. External tests correspond to interdependencies between components, like shared information or references from a component to another. All the tests aim to ensure the correctness of the CAP file and consistency of data contained in the CAP file.

3.2.1 Modelling each CAP file component

We have decided that it will be relatively easier to model the structural verifier as a syntactical analyser for CAP file. Therefore, each components constituting a CAP file is modelled. It includes the Header, the Directory, the Applet, the Import, the Class, the Constant Pool, the Method, the Static Field, the Export, the Reference Location, the Descriptor as specified in [20]. In figure 3, it corresponds to *component 1*, *component 2*, etc. At these standards components we add a specific custom component, i.e. the *Proof Component* that is relative to additional information used by the type verifier based on the Proof Carrying Code principles. For modelling purpose, each component has an associated B machine that contains properties on their respective content and services allowing to access their content. At the very abstract level, it is not necessary to represent all the details of a component but to provide a sufficient description of its properties and its services.
Each component is modelled as described in figure 4. It contains a set of abstract variables representing the state of the component, an invariant that defines properties over these variables, a specific operation for internal tests and a set of operations allowing to access to data contained within the component. The operation defining the

internal test modifies a boolean that indicates whether this component is correct or not (respectively TRUE or FALSE). This boolean is used as a pre-condition for all other operations. That is, we prove that an operation is executed only if the internal tests have been performed and succeeded.

```
MACHINE cpn_component

VARIABLES
            Set of variables used to describe the component,
            Component_verified

INVARIANT
            Set of properties on variables previously defined &
            Component_verified : BOOL

INITIALISATION
            Initialisation of all variables describing the component
            Component_verified := FALSE

OPERATION

            Res ← component_internal_verif=
            PRE Component_verified = FALSE
            THEN (Component_verified = TRUE => the component is correct)
            END;

            Res ← other_services_1=
            PRE Component_verified = TRUE
            THEN …
            END
            …
END
```

Figure.4. *Description of the model of one component of the CAP file*

The correctness of a component is expressed through its invariant. Internal tests are specified within the *component_internal_verif* operation. During the refinement process, more detailed information is added in order to implement the internal tests.
Note that, even if it is really important to have a formal specification of each component, it is not mandatory to have its formal implementation. In fact, as we are close to the CAP file format, it does not bring much to formally implement each component. We have decided to show that is possible to do so by formally implementing nine of the twelve components. But, errors found in formally implemented components and not formally implemented components are similar, both in terms of number and of origins. It mainly concerns errors due to the translation from informal specification to formal specification (wrong offset definition, lack of services, etc …). Designing the abstract machine helps to understand and clarify the informal specification. However, as one has to deal with low level implementation, it is hard to model and to implement efficiently. Moreover, the benefits are not as high as the cost for a formal development at this level.

3.2.2 Modelling test between component

The second part of the structural verifier is to perform what we refer to external test. It consists on ensuring that information shared or referenced by several components are coherent. To show this, one has to rely on the correctness of each component, independently. External tests are built on top of each component and their respective internal tests specification. Abstract machines representing external tests contain properties that must hold between components. To show the consistency between components, the model relies on two important points: properties of each component concerned by the test and services to access to data in order to compare them.
All external tests are modelled, refined and implemented in B. It is one of the main advantages of using formal methods as one can check the consistency. In fact, each component comes with a set of properties and a set of services. Formal specification allows to detect inconsistency between properties in different component and a possible lack of description. Formally implementing the external tests allows to detect a lack of services. This latter point concerns the operational part: we ensure that each component has all the services necessary for external tests implementation.
To model interdependencies between components, one has to use an informal document containing tests that has to be performed in order to ensure consistency between components. We have decided to associate to each component an abstract machine. This abstract machine includes all the external tests that have to be performed from this component to ensure its consistency among the others. Figure 5 describes such a machine.

The specification of external tests is simple. It is a translation of the informal specification into the formal specification. In the SEES clause of the B machine, one indicates all the components implied by tests described within this machine. It mainly contains the name of components interacting with the component being tested as well as some context machines providing definitions. There is no need to add any invariant since the proof relies on checking the consistency of components invariant and properties. So, the next and last B clause is the OPERATIONS one. Each operation describes one or more tests. The pre-conditions statement ensures that internal verification of each component concerned by the test has been performed. This allow the B prover to obtain more properties and hypothesis. The body of the operation contains the formal specification of the test, i.e. the abstract representation of what is performed by this test. In this representation, one uses abstract variables to express the property of the specified test. That is, the description of the relation between abstract variables of the different components involved by the test. The next step in the refinement process is the implementation. In the implementation, all abstract variables are replaced by concrete ones and access to data is performed by the different operations of each component.

```
MACHINE
        Cpn_component_ext

SEES
        All cpn_components concerned by the consistency of the component

OPERATIONS

        Res← test1=
        PRE Component_verified= TRUE &
             Component1_verified = TRUE &
             ...
        THEN Res :: bool(Description of the property)
        END

        RES ← test2=
        PRE...
        THEN ...
        END;

        ...
END
```

Figure.5. *Description of the specification of external tests for a CAP file component*

4 Integrating formal development into a smart card

4.1 Automatic code generation

4.1.1 B0 to C code translator

One of the main advantages of using B method is automatic code generation. In fact, at the end of the refinement process, the final component is an implementation in a subset of the B language, the B0. One of the main question when starting this formal development was about an efficient code generation. We have chosen to develop a simple code translator. The idea is to use it as a prototype to figure out what kind of improvements can be implemented and what kind of improvements are necessary. Note that improvement can be dedicated to a single smart card chip target but we do not consider this point here.

The translator that we used was developed within our laboratory. It is a very basic translator taking into account only implementation in B0 of the formal model. It translates B0 into C code. For an easier translation, we add types as assertion into the B0 code. It helps the translator choosing the best C type for the variable being translated. This allows, in particular, to restrict the variable memory space. For example, if one needs a boolean, he does not declare a 32-bits integer as a standard translator [5] does. In fact an integer requires 4 bytes whereas a boolean can be represented by a single byte.

As the B0 code is automatically translated, it seems not really important to optimise the B0 code. However, one aims to reduce the algorithm complexity of the implementation both by memory consumption and by execution time. No specific optimisation is taken into account. By specific, we mean specific to the smart card target chip.

To integrate C code into a smart card chip, one can think about strong optimisation on the C code translator. In fact, most of optimisations focus on the chip itself. Therefore, there is not a great need to optimise the B0 code as it can produce standard code. Then, depending on which chip target the program is compiled, one can use a translator especially optimised for that chip. The advantage here is to provide a B0 code that can be translated to several chip target without the needs to develop it for each chip. Once, a translator is designed for a specific chip, it can be used intensively.

4.1.2 Implementing into the ATMEL AT 90 platform

The smart card chip target is an ATMEL platform. First, one provides an implementation on a ATMEL AT90 SC 3232. This chip contains 32 kb for the software, 32 kb for the data and 1,5 kb of RAM. This implementation allows us to demonstrate the feasibility of embedding a verifier into a smart card. This first implementation embeds a complete type verifier excepting the *integer* management, and a partial structural verifier where external tests are excluded. This first implementation requires 29 kb of compiled code and fits in the ATMEL chip. The code of the verifier is stored in the first 32 kb for the software. Applets are loaded into the 32 kb dedicated to data.
The second implementation we provide is on an ATMEL AT90 6464 C with 64 kb for software, 64 kb for data and 3 kb of RAM. On this platform, one aims to embed the complete verifier, with the *integer* management for the type verifier and all the structural tests. With no specific optimisation, the size of this complete verifier is 48 kb.
The compilation chain provided by ATMEL which has some very efficient tools such as the compiler allows to gain a lot of space (in code size). Among the different options available for compiling the code for ATMEL, one chooses to focus on code size and not yet for code execution efficiency. Our goal is to restrict the code size and not necessarily the runtime efficiency, in order to embed the byte code verifier within a smart card. Later on, in section 3, results about execution of the verifier on some example applets are provided, showing that, even if the execution time can be improved, the first obtained results are not foolish.

4.2 Formal and informal development together into the smart card

The entire smart card is not modelled. Only a part of it, in our case the byte code verifier is modelled. A major question could be how far can we trust this development as not everything is formally developed. In this subsection we provide answers: Formal development has to be used when things are getting more and more complex. It helps improving quality of products and increasing trust. However, the smart card field is an area of specialists who are aware of specific issues and provides high quality code. Things are usually simple enough and well known to be handle by a specialist. Then using standard testing methods, one can avoid most of bugs. In fact, in smart card history, the use of formal methods was not necessary and the quality of products was very high.
With new applications, like the verifier, comes new complexity, hard to handle by man. Then, help of mathematics and others tools is mandatory to keep the same level of quality. Therefore, the development of verifier relies on basic blocks such as the memory management, the loading of the code. The formal code is then integrated to this pre-existent and well known part of the smart card. In order to be able to interact correctly an abstraction, a kind of interface, with these basic blocks is provided (Figure 7). It helps the formal development which needs information and properties about this blocks. It also helps defining accurately and with no ambiguity services required by the formal development and that have to be implemented by these blocks. Therefore, one has a gain both in terms of quality and of re-usability, as basic blocks are already existent.
Two major benefits appear with this integration: first re-usability of basic blocks and second a better confidence in basic blocks implementation. The former benefit is obvious since re-usability is a gain of time in a development. The latter one is a consequence of the modelling. If a developer spends time in designing a formal abstraction for his basic blocks, then he can know it better and can reason easily about it. Finally, errors can be discovered as well as lack of specification. All this leads to more confidence into the implementation.

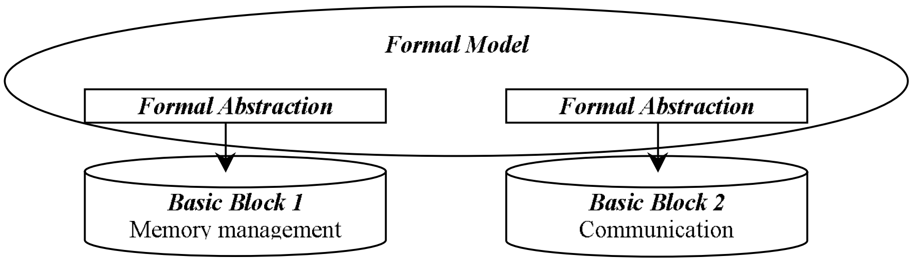

Figure.6. *Integrating Formal and informal development*

5 Metrics on the byte code verifier and its development

	Structural Verifier	Type Verifier	Utilities	Total
Number of lines of B	35000	20000	3500	58500
Number of components	116	34	45	195
Number of generated POs	11700	18160	950	30810
POs automatically proved (%)	81 %	70 %	77 %	74 %
Number of Basic machines	6	0	7	13
Number of lines of C code	7540	4250	860	12650
Workload (men months)	8	4	4	16

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

In this section, one provides metrics about the formal development of the byte code verifier. Table 1 synthesises metrics related to the development. In particular, one can note that the structural verifier is bigger than the type verifier. The reason is that the structural verifier contains a lot of tests, very different, that require a specification and an implementation for each one. Meanwhile, the type verifier can be seen as a single machinery 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 is two other results that are remarkable: the first one concerns the number of generated Proof Obligations (POs). The results shows that the type verifier generate much more POs than the structural verifier. The reason is that there are much more properties in the type verifier than in the structural verifier. The second results concerns the number of C code lines. This number is far less than the number of corresponding B code. The reason is that in the code translation, only implementation are taken into account. Moreover, INVARIANT clauses within implementations are not translated. This reduces drastically the number of lines translated from B to C.
One can then want to compare this development with others industrial B development (table 2). The one we have in mind is the subway control system designed by Matra Transport International, the Meteor project [2]. A first look to the table 2 allows us to say that project can be considered in complexity and in size as equivalent.

	Byte code verifer	Météor project
Number of lines of B	58500	115000
Number of generated lemmas	30810	27800
Percentage of automatic proof	74	81
Number of added rules	550	800
Atelier B version	3.6 PR 22	3.3

Table.2. *Comparing the byte code verifier and the Meteor project*

The first comment concerns the relation that exists between the number of lines of B and the number of generated lemmas. In fact, the byte code verifier requires less code than the meteor project whereas it generates more lemmas. The reason is that the type verifier generates a lots of POs, mainly because we have to embed into a single machine the 184 different cases. So, when the prover generates the lemmas, it takes into account these 184 different cases. Hence the high number of lemmas. Moreover, some particular instructions, such as the `invokespecial` one, requires to consider more than 40 different cases. Such instructions force the lemmas generator to generate as many as required lemmas. These lemmas are not really complicated: they are difficult to read but this is due to their length. In fact, in the byte code verifier, it is not possible to split more the machine that contains the 184 instructions specifications, as it can be the case in the meteor project. The split should have as consequence to reduce the number of generated lemmas. The second important point concerns the number of added rules. In both project, this number is roughly similar. It means that to develop a B project, it is necessary to also develop a set of added rules that aim to ease the proof process. Adding rules to the Atelier B is important in the proof activity as it can speed up this step.

6 Conclusion

Adding an embedded byte code verifier to a Java Card allows the card to ensure its own security. It is important when one thinks about deployment architecture. Today, post-issuance on smart card, i.e downloading new code when the card is already on the field, requires an heavy infrastructure which implies cryptographic protocols and certification center. With an on-card verifier, the deployment infrastructure is light as the card ensures its own security.

Now an implementation of a Java Card byte code verifier is available, one can follow different paths: the implementation can be improved and optimised in order to integrate it with a virtual machine. This integration is not simple and must take into account several parameters such as the load and the link of the CAP file inside the card, as well as information storage within the card. A different path, but not orthogonal to the previous one, can be to add some cryptographic features to this existing prototype. This would provide a tamper-proof verifier that could be used for signing applets.

The proof process applied during the formal development of the byte code verifier ensures its correctness. Moreover, we provide a clear and complete specification of what a verifier should do. A positive points is that one has specified all the tests with no optimisation. Then, our prototype can be used as a on-card reference implementation. These are the major benefits one can bring into the fore on the use of formal methods. Another important results is the knowledge acquired to integrate a formal development into existing traditional development as well as the cost of a formal development and the cost of the integration. Using formal methods is not free. A trade-off has to be found in order to obtain the best confidence into the code at the lowest cost. For example, we show that the formal development of each CAP file components is not necessary. The reason is that errors found in formal development and traditional development are similar. Therefore, one should not spend time and money on this development. However, we also showed that a formal description of each component is mandatory not only for the remaining formal development but also to detect a lack of consistency of each component specification.

7 References

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

[2] Behm P., Benoit P., Faivre A., Meynadier J. –M., Météor: A Successful Application of B in a Large Project, *Proceedings of FM'99-Formal Methods*, Volume 1, LNCS 1708, pp.369-387, Toulouse, France, September 1999.

[3] Y. Bertot, *A Coq formalization of a Type Checker for Object Initialization in the Java Virtual Machine*, Research Report, INRIA Sophia Antipolis

[4] 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.

[5] L. Casset, *Formal Implementation of a Verification Algorithm Using the B Method*, Proceedings of AFADL01, Nancy, France, June 2001

[6] ClearSY web site, http://www.clearsy.com

[7] The Coq Proof Assistant web site, http://coq.inria.fr

[8] Isabelle web site, http://www.cl.cam.ac.uk/Research/HVG/Isabelle/index.html

[9] G. Klein, T. Nipkow, *Verified Lightweight Bytecode Verification,* in ECOOP 2000 Workshop on Formal Techniques for Java Programs, pp. 35-42, Cannes, June 2000.

[10] X. Leroy, *On-Card Byte Code Verification for Java Card*, Proceedings of e-Smart, Cannes, France, September 2001.

[11] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification,* Addison Wesley, 1996

[12] G. Necula, P. Lee, Proof-Carrying Code, in 24th 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/popl97.ps.gz

[13] T. Nipkow, *Verified Byte code Verifiers*, Fakultät für Informatik, Technische Universität München, 2000. http://www.in.tum.de/~nipkow

[14] C. Pusch, *Proving the Soundness of a Java Bytecode Verifier in Isabelle/HOL*, In OOPSLA'98 Workshop Formal Underpinnings of Java, 1998.

[15] C. Pusch, T. Nipkow, D. von Oheimb, *microJava: Embedding a Programming Language in a Theorem Prover*. In Foundations of Secure Computation, IOS Press, 2000.

[16] 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.

[17] Z. Qian, A. Coglio 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.

[18] 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.

[19] 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

[20] *Java Card 2.1.1 Virtual Machine Specification*, Sun Microsystem, 2000.

[21] *Connected, Limited Device Configuration*, Specification 1.0a, Java 2 Platform Micro Edition, Sun Microsystems, 2000.

[22] Trusted Logic's web site
http://www.trusted-logic.fr/