FACADE: a Typed Intermediate Language Dedicated to Smart Cards

Gilles Grimaud[1], Jean-Louis Lanet[2], and Jean-Jacques Vandewalle[2]

[1] Université de Lille, LIFL/RD2P,
Gilles.Grimaud@lifl.fr, http://www.lifl.fr/~grimaud/
[2] Gemplus Research Lab,
Jean-Louis.Lanet@gemplus.com, and jeanjac@research.gemplus.com

Abstract. The use of smart cards to run software modules on demand has become a major business concern for application issuers. Such downloadable executable content requires to be trusted by the card execution environment in order to ensure that an instruction on a memory area is compliant with the definition of the data stored in this area (*i.e.* its type). Current solutions for smart cards rely on three techniques: for Java Card, (1) an off-card verifier-converter performs a static verification of type-safety, or (2) a defensive virtual machine performs the verification at runtime; for others, (3) no type-checking is carried out and the trust is only based on the containment of applications. Static verification is more efficient and flexible than dynamic techniques. Nevertheless, as the Java verifier cannot fit into a card, the trust is dependent on an external third-party. In this way, the card security has been partly turned to the outside. We propose and describe the *FACADE* language for which the type-safety verification can be performed statically on-card. In this paper, we focus on the verification process for which a formalization is given. A model in the B language exhibits the type-safety properties to be checked. This work is aimed at proving the correctness of the verification.

1 Introduction

In this section the specific domain of smart cards is described. For people not aware about smart cards, we briefly review the technology and the history of smart cards from embedded devices dedicated to specific applications up to open platforms for enabling the downloading of new services all along the card's life.

1.1 The Specific Domain of Smart Cards

Smart cards form a specific domain by three ways we detail hereafter: their internal constitution, their interfaces, and their applications.

A smart card is a piece of plastic, the size of a credit card, in which a single chip microcontroller is embedded. Usually, microcontrollers for cards contain a microprocessor (8-bit ones are the most widespread, but 16-bit and 32-bit

processors can now be included in the chip) and different kinds of memories: RAM (for run-time data), ROM (in which the operating system and the basic applications are stored), and EEPROM (in which the persistent data are stored). Since there are strong size constraints on the chip, the amounts of memory are small. Most smart cards sold today have at most 512 bytes of RAM, 32 KB of ROM, and 16 KB of EEPROM. This chip usually also contains some sensors (like light sensors, heat sensors, voltage sensors, *etc.*), which are used to deactivate the card when it is somehow physically attacked.

In order to be usable, a smart card must be inserted in a card reader/writer, which provides power to the card, as well as a clock signal. Also, any communication between the terminal and the card goes through the card reader/writer in the form of messages exchanged from the terminal to the card (commands), and respectively, from the card to the terminal (responses). All these basic aspects are strongly standardized, since cards are meant to be usable with a wide range of devices. The family of ISO 7816 standards are the references [3]. They standardize many features, from the positions of the contacts on the card to the transport protocol that is used to communicate between the card and the card reader/writer, as well as the format of the messages exchanged between the terminal and the card.

A smart card can be viewed as a "data safe", since it stores data in a secure manner and it is used securely during short transactions. Its hardware is the base for its safety. The fact that the chip in a card is embedded with sensors in plastic and glue, and that all components are on the same chip makes a physical attack quite difficult. The software is the second barrier for its safety. The chip programs are usually designed for neither returning nor modifying sensitive information prior to be sure that the operation is authorized. In fact, most card applications use the card either to safely store data, or to process sensitive data. Most smart cards include some support for cryptographic functions, which allows them to secure their transactions with the outside world. More practically, cards are often used either to manage some kind of currency (money or tokens) or to identify a person.

1.2 Smart Cards State-of-the-Art

The specific domain of smart cards is close to the domain of embedded devices. Like embedded devices, smart cards are aimed toward the consumer electronics market, which requires from these systems even more and more convenience and flexibility.

The methods, languages and tools for developing a smart card system share some characteristics with those of the embedded domain. Until recently, smart card codes were written in hand coded native assembler. All programs (drivers, operating system, libraries, applications) were developed in a monolithic piece of code burned in the ROM of the smart card. Therefore, not only traditional card systems are difficult to develop (low-level programming language, very reduced-feature microcontroller, specific code for every microprocessor) but also they

cannot support evolution of their applications since all the application code is burned forever with the runtime engine in the ROM.

Moreover, the production of such a "petrified" monolithic program dedicated to a specific hardware and with *ad hoc* functions for the application domain consumes most of the card development cycle. In order to issue a card application, it is needed (i) to write precise specifications, (ii) to write or re-write the basic software (akin to an operating system) for possibly multiple platforms, (iii) to develop specific functions for the application, and (iv) to verify this software before to deploy it on thousands or millions of cards. This process is time-consuming and costly. Since defining specifications for products that will be available long after is risky, this process is a difficulty for the creation of new markets. As it requires a long time it also severely limits the ability of a card issuer to deploy rapidly new applications in accordance with the market needs.

In order to cope with these market needs (to simplify, a reduced "time-to-market" and flexibility for card applications) new generations of smart cards (called *open* smart cards) have emerged during the last two years. Most notable efforts towards such smart card systems are Java Card [14, 15], MultOS [6], and Smart Card for Windows [7] which provide application developers an opportunity to create applications on a common base of code. They contain a platform for the dynamic (*i.e.* on demand) storage and the execution of downloaded executable content, which is based on a virtual machine for portability across multiple smart card microcontrollers and for security reasons.

While these new smart cards bring solutions regarding the market needs, they also introduce new problems for smart card makers. They provide solutions for card application developers by enabling them to program in high-level languages, on a common base of software (an abstract machine and application programming interfaces) which isolates their code from specific hardware and operating system libraries. In that sense, they can reduce drastically the time to get new applications to market. They also tend to support both the flexibility and the evolution of applications by enabling the downloading of executable content in already deployed smart cards. This later characteristic requires more sophistication in term of security techniques since any program (particularly, attention must be brought to on malicious or erroneous ones) could potentially damage or misappropriate the whole card system. This paper deals with this issue as summarized in the next section.

1.3 Outline of this Paper

Ensuring that a program cannot damage a system consists in denying it access to other memory areas than those reserved for its execution and data (containment or "sandboxing"). Containment relies on access controls to memory areas. It would be better to associate containment with a protection mechanism that also verifies that every instruction accesses data with respect to their types. In fact, a more fine-grained protection consists in verifying that every instruction that accesses to a memory area is compliant with the definition of the data stored in this area (*i.e.* its type, or its class in an object-oriented system). This later

technique has been popularized in the Java language [5] with the verification step performed by the virtual machine each time it loads a new class from an unsecure source.

In the field of open smart cards, this security issue is generally addressed by the use of three techniques: for Java Card, (1) an off-card verifier-converter performs a static verification of type-safety, or (2) a defensive virtual machine performs the verification at runtime; for others, (3) no type-checking is carried out and the trust is only based on the containment of applications. For traditional smart cards (with all the code burned in their ROM), these techniques are not used because the confidence in the programs is based on the fact that they have been tested and verified before the delivery of the cards.

In this paper, we propose a new architecture to build smart card code that is based on a typed intermediate language called FACADE. It aims at providing a coherent system for card software engineering enabling both, the production of efficient traditional smart cards, and the building of type-safe downloadable content for open smart cards.

Section 2 debates the security schemes used in open smart cards before detailing the FACADE architecture. In Section 3 we focus on the verification process for which a formalization is given. A formal model using the B language exhibits the type-safety properties to be checked. Finally, Section 4 summarizes the paper and gives our forthcoming future work.

2 FACADE Architecture

2.1 Language Related Work

The current open smart cards provide programmers with the ability to download programs dynamically. This characteristic aims at making smart card systems more convenient in two ways. On the one hand, they support the use of high-level languages (like Java with Java card, or Visual Basic with Smart Card for Windows) in spite of smart card constraints. On the other hand, they guarantee that the programs loaded in a smart card are not able to compromise the security of the other programs.

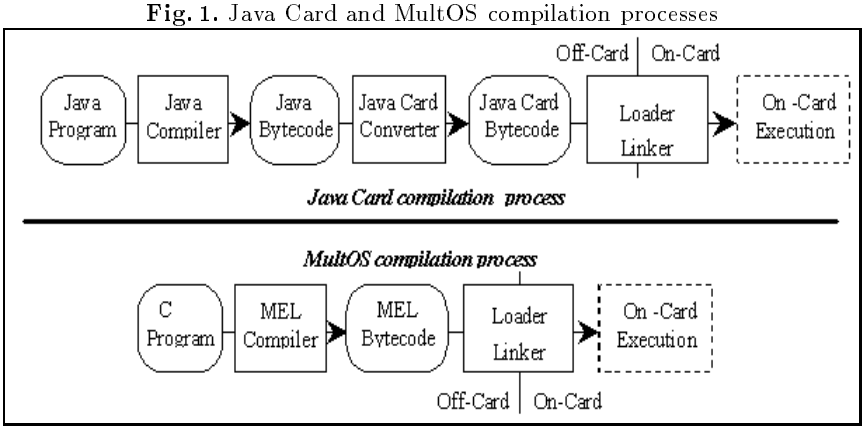
Smart Card Compilation and Loading Process Though current smart cards are able to run programs written in high-level languages, they still have drastic limitations. Generally, the compilation of programs expressed in high-level languages produces a code unsuited to the available hardware. Two solutions have been used in the existing products.

The first solution consists in defining programming languages dedicated to smart cards. These languages tend to be close to those used on workstations but, they force programmers to take into account some specific aspects of smart card microcontrollers. For example, the language MEL has been specifically defined to program the MultOS card operating system [6]. For convenience, a domain specific compiler can generate MEL code from C.

Another solution consists in converting the code produced by a traditional compiler into a specific code adapted to the smart card constraints. For example, the Java class file format is too heavy to be loaded in Java Card. Thus, the class files produced by compilers will be treated by a specific converter in order to get a compressed version of class files [16]. Smart Card for Windows is another example of the same technique.

Figure 1 presents the process of compilation and loading associated with these two approaches.

Fig. 1. Java Card and MultOS compilation processes

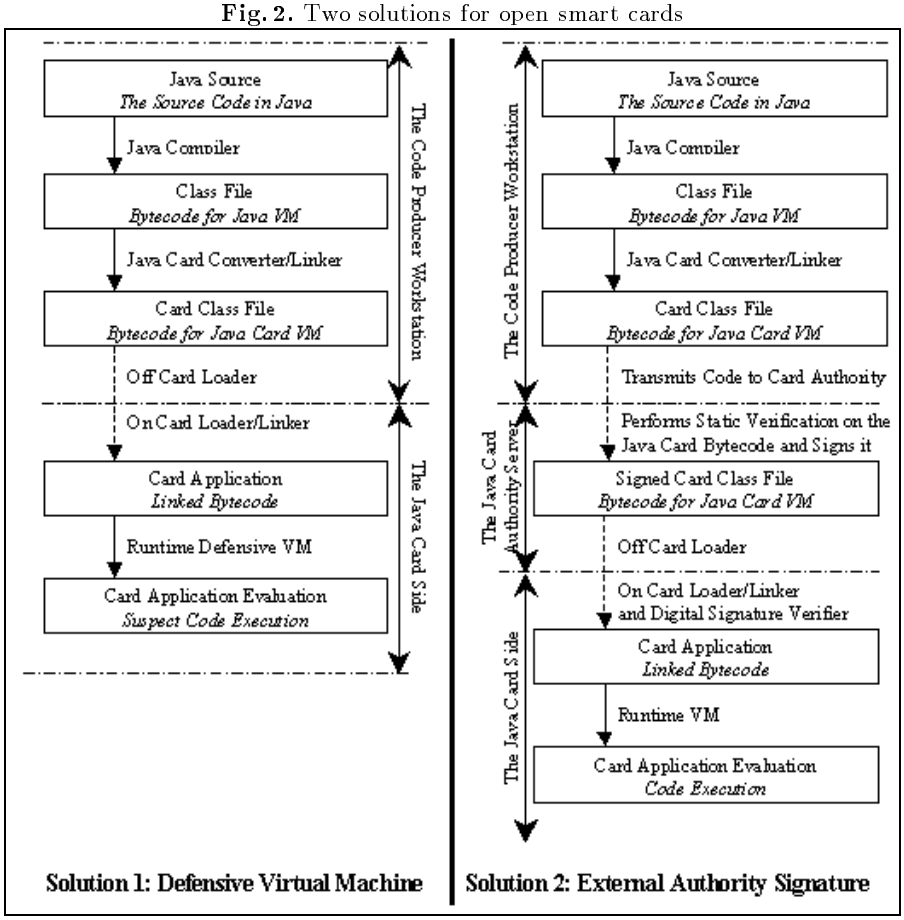


Code Downloading and Smart Card Security Currently, the Java Card framework does not support Java-like dynamic class loading. The main problem is that a smart card environment is too small for running the Java class file verifier. Open card architectures hence propose a downloading framework with reduced flexibility, in which the download unit is the application (or package in Java) rather than a single class file. In addition, new frameworks for program distribution and downloading have been proposed. The two major proposals are outlined in Figure 2.

The first solution is known as *defensive virtual machine*. All safety checks are performed at run-time, hence replacing the pro-active static bytecode verification by a defensive run-time bytecode verification. For instance, before to perform a write operation to a given memory area, the virtual machine checks the type and access rights associated with the data located in that target area. In Java Card, the checks are mostly related to typing; in MultOS, the checks are related to access right checks, since security is based on application containment.

Defensive virtual machines have two major drawbacks. First, the number of run-time checks to be performed severely hinders their performance; then, some

Fig. 2. Two solutions for open smart cards



additional data (such as typing information) needs to be stored, which increases the memory requirements. As a consequence, this technique can only be applied to small programs, and this drastically reduces its flexibility and usefulness.

The second solution consists in performing the security checks outside the smart card. Before to download a program onto a smart card, it is first transmitted to a trusted third-party (usually the card issuer), which checks the program and certifies it. When the program is downloaded, it is accompanied with its certificate. Before to allow this program to run, the smart card checks the validity of the certificate. This solution is currently the one that is proposed by the major card issuers. However, it has a severe drawback: the whole security of the smart card system relies on the security of the certification scheme. This solution is satisfactory for local deployments, but it can be unsatisfactory for long-term deployment of large scale applications. In fact, a central point of trust (the certification authority) creates a single point of failure that can compromise all the

security architecture in case of attack. Also, in some cases, this centralization is not well suited to application needs in which the various participants are not able to trust a same authority. Moreover, the implementation and deployment of a certification infrastructure is difficult and requires costly operations like monitoring and maintenance. For these reasons, we argue that it would be far preferable to design a solution in which the security of the smart card system relies on the smart card system rather than on a certification scheme.

Another Approach: FACADE A recent research work [12] proposes to adapt the Proof Carrying Code [9] technique as a mean for verifying on-card the type-safety of Java Card programs. This technique seems very promising, but the complexity of the on-card processing for a full type-checking makes it still difficult to implement on current smart cards. We propose to solve this problem by introducing a typed intermediate language called FACADE in which the type-safety can be verified on-card. As, this language is intended to be a target platform for multiple high-level languages such as Java or Visual Basic, it also solves some interoperability problems between new open smart card systems. The FACADE language is also a part of a framework for producing card programs. Inside this framework, it has two main properties: (1) it is designed specifically for an implementation on reduced feature devices, and (2) it is designed to enable an efficient static checking.

2.2 The FACADE System

The FACADE language is targeted toward very small platforms, but it is not specifically targeted toward open smart cards. The advantage of using such an intermediate language is that it can be used for all kinds of smart cards and small devices. The language is secure enough to be downloaded into an open smart card, but it can also be used to produce optimized native code, to be included in the ROM of a smart card. The main application of such an idea would be to use an high-end open smart card to prototype an application, and then to produce an optimized dedicated code in order to obtain smaller (and cheaper) cards for mass production.

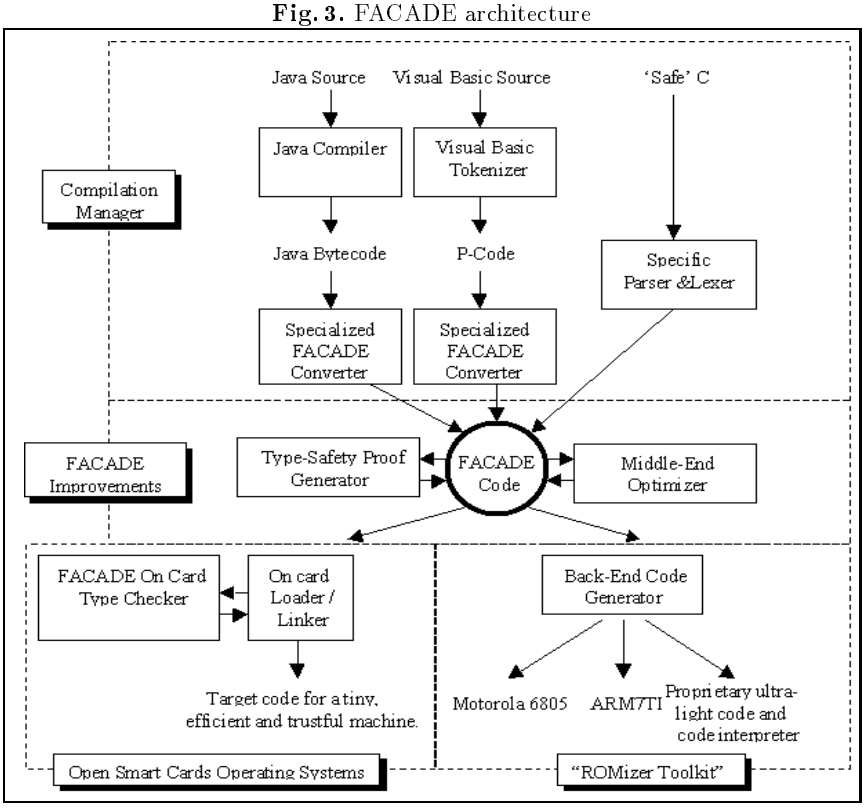
In our system, the FACADE typed intermediate language is a central element, as shown in Figure 3. Programs written in various source languages are first fed into a language-specific *off-card front-end* which does lexical analysis, parsing, type-checking, and compilation into specific representations. Then, it is translated into the FACADE intermediate format. The *off-card middle end* does conventional optimizations. It also generates the elements that will be necessary to prove the type-safety of the program when it will be loaded into the smart card. At this step, the FACADE code may be used in two different directions:

1. The *on-card back end* for open smart cards verifies the type-safety of the program at loading time, and generates a dedicated code for the target hardware or software machine.

2. The *off-card back end* for *ad hoc* smart cards used the FACADE code as the source code for producing a complete card code dedicated to be burned in ROM thanks to the usage of a ROMizer[1]

All the production tools are deliberately made independent of each other so that they may support different source languages and different target platforms. This paper only deals with the proof generator and verifier tools. Other components of this architecture are currently investigated by ongoing experimental research works.

Fig. 3. FACADE architecture



Using a common intermediate language to share compiler infrastructure is not a new idea. Many compilers have used or use a common intermediate format for multiple source languages (*e.g.*, GNU gcc, Eiffel, Pascal, *etc.*). For a long time, Eiffel has used the C language as its intermediate language. For more

[1] A ROMizer is a software used to produce ROM binary images of programs.

advanced systems, specific languages have been defined. For instance, the FLINT architecture [13] is based on a typed intermediate format that supports higher-order functions and an advanced polymorphic type system. However, none of these languages is appropriate for small platforms such as smart cards.

The main characteristic of FACADE is its ability to prove the type-safety of FACADE code using small hardware like a smart card. This feature is very important in order to use the language as a target for the compilation of the Java language, whose security relies heavily on type-checking. Here, instead of trying to fit a standard Java type-checker on a smart card (as done in [11]), we have chosen to design a language which caters to the very specific needs of smart cards. In the following sections, we focus on the way in which type-checking can be performed on the FACADE language on a smart card.

2.3 Principle of Type-Checking: Type Inference

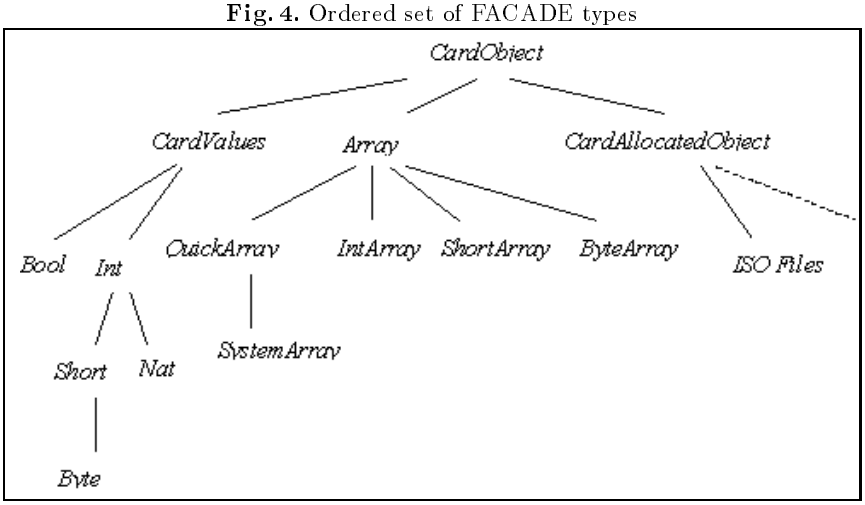
Program checks are usually based on data flow analysis. This analysis aims at determining the type of the variables at each program point (*pp*). The type of some variables changes during the execution of the processing because they are temporarily used for different computations.

The Hierarchy of Types In the FACADE architecture, all information are associated with types. FACADE is an object-oriented intermediate language. Thus, types are commonly defined by classes. The hierarchy of classes is an extensible data structure. Indeed, when new applications are loaded, they add their own classes to those already recorded in the card system. These extensions are done in a tree-based model. Each class has one and only one super class. Moreover, one particular class: $CardObject$ is the top of the hierarchy. Thus, directly or indirectly, any class extends $CardObject$, which is the root of the inheritance graph.

For the data flow analysis, we define a semi-lattice as a tuple $L = (V, \subseteq, \cap)$ where V is the set of the class types, $\subseteq$ is a partial order defined over V, and $\cap$ is a binary operation defined over V. Two elements $i, j \in V$ are incomparable, if neither $i \subseteq j$ nor $j \subseteq i$. We say that if $i, j \in V$, j covers i if $i \subset j$ and there is no k such that $i \subset k \subset j$. In the diagram of an ordered set $(V, \subseteq)$, two elements $i, j \in V$ are directly connected if one covers the other. The Figure 4 presents the ordered set of FACADE types. The greatest element $CardObject$ of V is called the top element of V and can be written $\top$.

Type-Safe Operations A program is made of a sequence of elementary operations. Each one uses some number of input data and modifies some number of output data. In FACADE, there are only five distinct elementary operations. Following, for each of them we informally provide (i) their operational semantics, and (ii) their static semantics.

Fig. 4. Ordered set of FACADE types



1. `Return` $VarRes$
   (i) This operation ends the execution of the processing and returns the value of the variable $VarRes$.
   (ii) This operation is legitimate if the type of the returned variable ($VarRes$) is a subtype or equals of/to the return type declared in the method signature.
2. `Jump` $LabelId$
   (i) This operation is an unconditional jump to another operation of the program marked with the label $labelId$ defined in a label table.
   (ii) This operation is defined only if the label $labelId$ exists and the value of the associated pp is valid.
3. `Jumpif` $Var\ LabelId$
   (i) This operation is a conditional jump. The Var variable is of type $Bool$ which determines if the jump must take place or if it must be ignored.
   (ii) If the label $labelId$ exists and the variable Var is of type $Bool$, this operation is correct.
4. `Jumplist` $Var\ nbcase\ \{LabelId1, LabelId2 \ldots\}$
   (i) This conditional jump reaches the label whose sequence number, in the list of the labels, is equal to the variable Var. If the variable Var is a number greater than the cardinal of the label list (defined by the immediate value $nbcase$), the operation does not perform any jump but increments the program point.
   (ii) If each label $LabelId_x$ exists, if the number of labels is less or equal than $nbcase$, and if the variable Var is of type Int, then the operation can be carried out.
5. `Invoke` $VarRes\ Var\ methodId\ \{tabVar1, tabVar2 \ldots\}$
   (i) This operation executes the method $methodId$ on the variable Var, with the parameters $tabVar1, tabVar2 \ldots$ $VarRes$ is the variable that contains

the value returned by the method call. In fact, this operation of the intermediate language FACADE could be translated in the card, according to the nature of the method $methodId$, either like a jump into another procedure, or like a set of elementary operations of the target machine.
(ii) The call of a method is type-safe if first (a) the method $methodId$ is declared in the class (the type) of Var, and second (b) the types of the variables $tabVar1, tabVar2 \ldots$ are those awaited by the method $methodId$ of the class of Var, and finally (c) the type of the variable $VarRes$ is that turned over by the method $methodId$.

An elementary FACADE operation is considered type-safe if the inferred types are conform with those defined by the static semantics. For instance, let `i` and `j` be respectively of types Int and $IntArray$, an incorrect operation can be `Invoke j,i,add,j`. However, the addition method of the Int class takes an integer value as the second parameter. As $IntArray$ is not a subtype of Int, such operation is illegal. In this other example, let i a variable of the type $Short$, the following operation is correct: `Jumplist i, 4, (l1, l2, l3, l4)`, because the type $Short$ is a subtype of Int.

In fact, the variable types check may appear simple if the type of a variable do not change during the life span of the method. Unfortunately, it is not the case for all kinds of variables. We distinguish two kinds of variables, those which have an invariable type (*e.g.*, the attributes of an object), and those which have a variable type (*e.g*, the variables used for temporary storage during the evaluation of complex expressions). In the first case we speak about $Local$ and in the second, we use the term Tmp.

Type-Safe Verifier and Smart Card Constraints A type verifier proves that each instruction contained in a program will be executed with values having the types expected by the instruction. The type verifier reads the program and notes for each elementary operation, the produced types and the consumed types.

Generally the execution of a program is not sequential. This is why a traditional verifier follows the arborescent path of each case of the program execution. The computation of this tree structure establishes with certainty the types of the variables consumed and modified by the elementary operations. When various paths end in a same program point, the safe-type verifier determines a single type for each Tmp variable.

The fixpoint search is the most complex aspect of the type-checking algorithm. As shown previously, it requires the production of a complex data structure, and thus the execution of a complex recursive processing. The smart card cannot perform this recursive processing, because of its hardware limitations. Therefore, no smart card proposes an operating system performing at load time type-safety checks. And if the type-safety checks are performed at runtime, they slow the virtual machine.

A Two-Step Solution Some studies propose to distribute the type-checking of a downloadable code between a code-producer and a code-receiver. The most outstanding works in this field are Proof Carrying Code [9] (PCC) and Typed Assembly Language [8] (TAL). The main idea of these works is to generate by the code-producer a proof of the program safety. The code is transmitted with this proof to the receiver. The code-receiver checks the program using the proof. The interest of this technique is that the proof verification is much easier than the proof production.

In our case, the proof is the result of the type combination for a given label. Like TAL, we propose to provide, with the transmitted program, a list of labels with a state of the *Tmp* variables for each of them *i.e.* for each program point which can be reached by a jump. The receiver performs a sequential analysis of the code. When an operation is marked by a label, the receiver checks if its current inferred types are lower or equal (relation $\subseteq$ of the types hierarchy) to those defined by the proof. Moreover, when the receiver checks a kind of jump, it makes sure that its inferred types are lower or equal to those defined for the label. A program is refused if a control fails. Thus the receiver makes sure that the program is safe, but it does not perform any complex algorithmic operation. We formalize this type-checking process in the third part.

Using Formal Methods In our architecture, the card security relies mainly on the verification process. Therefore, the design of the type-checker cannot suffer any error. For that, we design a formal model of our embedded type-checker. The goal is to provide a formal proof of its implementation. The useful elements of the formalization are presented in the next section.

3 Type-Safety Verification

3.1 Description of the Verifier

In the FACADE approach, the verification process is split in two parts. The resource consuming algorithm (*i.e.* the label generator) is done on the terminal side (code-producer) while the verification of the labels and the type inference between two labels is done on the card side (code-receiver). The first algorithm is a traditional type reconstruction and verification of all the possible execution paths. For each label (*i.e.* each point that can be reached by a jump operation), the joint operation is computed for the untyped local variables. The label table is transmitted to the card with the FACADE code. At loading time, the card computes sequentially the inference and compares it with the label table. In case of divergence the card refuses the code.

The verification process must ensure that every execution will be safe. It means that starting in a safe state each operation will lead the system in another safe state. For that purpose, a transition system describing a set of constraints is constructed. It defines a correct state for each operation (preconditions) and how

the system state evolves (post-conditions). The both preconditions and post-conditions define the static semantics of FACADE. In particular, it must be ensured that:

- all instructions have their arguments with the right type before execution,
- all instructions transform correctly the local variables for the next pp.

Static Semantics of FACADE The system maintains some tables: the class descriptors ($classDsc$) and the method descriptors ($methodDsc$). The class descriptors table contains the definitions of the classes present in the card and the class hierarchy. The method descriptors table contains for every class the signatures of the already checked methods. Once a method has been accepted by the verifier, it is added in the method descriptors.

We briefly introduce our notation. Programs are treated as partial maps from addresses to instructions. If Pgm is a map, $Dom(Pgm)$ is the domain of Pgm. $\forall pp \in Dom(Pgm), Pgm_{pp}$ is the value of Pgm at program point pp, which is written $Pgm[pp \rightarrow v]$. A program P is a map from program points to instructions. The model of our system is represented by a tuple: $< pp, Local, Tmp >$ where pp denotes a program point (or program counter), $Local$ the set of typed local variables, and Tmp the set of untyped local variables. The vector of map T and the map L contain static information about the local variables. The vector T assigns types for the Tmp variables at program point pp such that $\forall i, (Tmp[i] : T_{pp}[i])$. The map L assigns types for the $Local$ variables, which are never changed, such that $\forall i, (Local[i] : L[i])$. A program is well typed if there exist L and T that satisfy: $L, T \vdash P$.

Following (see Figure 5) is given a formal definition for the control flow instructions (`Return`, `Jump`, `JumpIf`, and `JumpList`) and for the invocation instruction (`Invoke`). There are two static semantics for the `Invoke` instruction depending on the kind of variable used for $VarRes$.

Off-card Label Generation During this phase, we can construct the type information for local variables for each program point using a traditional type inference algorithm (for example, the Dwyer's algorithm [2]) and then, we calculate for every label the value of T for every path. An inference point (ip) is a couple made of a program point associated with the T type table for that program point (written T_{extern}). They are collected in a table – the $labelDsc$ table – which is sent to the smart card. When a new method has been loaded, it is verified that the label descriptors only reference valid addresses: $\forall i, (i \in labelDsc[pp], i \in Dom(P))$. Finally, we can formally define the label descriptors by:

$$\begin{gathered} ip \in labelDsc, ip = (pp, T_{extern}) \\ pp \in Dom(P) \\ T_{extern} = T_{pp} \end{gathered}$$

Fig. 5. Static semantics of FACADE control flow and invocation instructions

$$\frac{\begin{array}{c} P[i] = \texttt{Return}\ VarRes \\ i \neq Card(P) \\ VarRes : methodDsc[ThisMethodId][ReturnType] \\ i+1 \in Dom(P) \end{array}}{L, T, i \vdash P}$$

$$\frac{\begin{array}{c} P[i] = \texttt{Jump}\ LabelId \\ LabelId \in Dom(P) \end{array}}{L, T, i \vdash P} \qquad \frac{\begin{array}{c} P[i] = \texttt{JumpIf}\ Var\ LabelId \\ Var : Bool \\ LabelId \in Dom(P) \\ i+1 \in Dom(P) \end{array}}{L, T, i \vdash P}$$

$$\frac{\begin{array}{c} P[i] = \texttt{JumpList}\ Var\ nbcase\ Label \\ Var : Int \\ nbcase : Nat \\ nbcase \geq Card(Label) \\ \forall\, pp, (pp \in Label \Rightarrow pp \in Dom(P)) \\ i+1 \in Dom(P) \end{array}}{L, T, i \vdash P}$$

$$\frac{\begin{array}{c} P[i] = \texttt{Invoke}\ VarRes\ Var\ MethId\ tabVar \\ MethId : methodDsc_{classDsc(Var)} \\ VarRes \in T \\ T_{i+1} = T_i[VarRes \rightarrow methodDsc[MethodId][ReturnType]] \\ \forall\, i, (i \in 1..methodDsc[MethodId][nbvar] \Rightarrow tabVar[i] : methodDsc[MethodId][i]) \\ i+1 \in Dom(P) \end{array}}{L, T, i \vdash P}$$

$$\frac{\begin{array}{c} P[i] = \texttt{Invoke}\ VarRes\ Var\ MethId\ tabVar \\ MethId : methodDsc_{classDsc(Var)} \\ T_{i+1} = T_i \\ VarRes \in L \\ VarRes : methodDsc[MethodId][ReturnType] \\ \forall\, i, (i \in 1..methodDsc[MethodId][nbvar] \Rightarrow tabVar[i] : methodDsc[MethodId][i]) \\ i+1 \in Dom(P) \end{array}}{L, T, i \vdash P}$$

Consider the following example (see Table 1), this piece of FACADE code (we call it program P) uses two variables i and v in order to compute a loop from 0 to 100.

Table 1. A sample FACADE program

pp	Label	Instruction	Comment
1		`Invoke i, 0, likeIt`	`i` is set to 0
2		`Jump label0`	go to the test at `Label0`
3	`Label1`	`Invoke i, i, add, 1`	increment `i`
4	`Label0`	`Invoke v, i, LtOrEq, 100`	test if `i` less or equal than 100
5		`JumpIf v, label1`	if TRUE go to `Label1`
6		`Return void`	otherwise, return

Using a fixpoint algorithm, the inference procedure checks all the paths of the tree. In this example, it needs eight steps as illustrated Table 2.

Table 2. Off-card type inference procedure of the sample FACADE program

Step	*pp*	Label	Instruction	$T[i, v]$
1	1		`Invoke i, 0, likeIt`	$\{\top, \top\}$
2	2		`Jump Label0`	$\{Int, \top\}$
3	4	`Label0`	`Invoke v, i, LtOrEq, 100`	$\{Int, \top\}$
4	5		`Jumpif v, label1`	$\{Int, Bool\}$
5	3	`Label1`	`Invoke i, i, add, 1`	$\{Int, Bool\}$
6	4	`Label0`	`Invoke v, i, LtOrEq, 100`	$\{Int, Bool\}$
7	5		`JumpIf v, label1`	$\{Int, Bool\}$
8	6		`Return void`	$\{Int, Bool\}$

At step 1, i and v are Tmp variables for which the types are unknown before the operation. The type of the return variable for the method `likeIt` from the class Int is Int. Thus, at step 2, i is assigned the type Int. At step 4, a first choice is made by jumping at *pp* 3, the other choice is verified at step 8. At step 6, the state is different than the step 3 (for the same *pp*) thus, we have not reached a fixpoint. At step 7, states are identical, meaning that a fixpoint has been reached. Now, the pending path memorized at step 4 can be checked.

For the *pp* 3, we have to make the combination between the two states $T_4[i, v] = (Int, Bool)$ (at step 6) and $T_4'[i, v] = (Int, \top)$ (at step 3). Using our semi-lattice, the greatest lower bound of $T_4[i, v]$ and $T_4'[i, v]$ is $T_{extern}[i, v] = (Int, \top)$. Our *labelDsc* is made of two *ip*: $(4, [Int, \top])$ and $(3, [Int, Bool])$.

In order to optimize the size of the data sent to the smart card, the *labelDsc* table can be reduced. In fact, by verifying the definition-use paths, one can remark that the information on the variable v at step 6 is useless because v is always defined before any use. Thus, it is possible to simplify the *labelDsc* as shown Table 3.

Table 3. Simplification of the label descriptors table (*labelDsc*)

ip	*pp*	$T_{extern}[i, v]$
0	4	$\{Int, \top\}$
1	3	$\{Int, \top\}$

On-card Type Inference and Verification The on-card FACADE verifier uses the off-card-generated label descriptors in order to reduce its footprint in memory as well as the time needed by this process. After receiving the application and its *labelDsc*, the verifier checks the conformity of the label descriptors. Then, it begins the verification process by running sequentially through the code. The first advantage is that it requires only six steps instead of eight for the verification process. The second advantage is linked to the used space: on the smart card we have to maintain only the current Tmp type T_c, instead of the complete vector of maps T.

The procedure is the following (see Table 4): for each instruction the verifier checks if the program point is referenced into the label descriptor. In such a case, the table T_{extern} replaces T_c otherwise, the algorithm uses T_c. It applies the static semantics to verify the correctness of the code. After an unconditional `Jump`, the following instruction must have necessarily a label, and then it replaces T_c by T_{extern}.

Table 4. On-card verification procedure of the sample FACADE program

Step	*pp*	Instruction	$T_{extern}[i, v]$	$T_c[i, v]$
1	1	`Invoke i, 0, likeIt`		$\{\top, \top\}$
2	2	`Jump label0`		$\{Int, \top\}$
3	3	`Invoke i, i, add, 1`	$\{Int, \top\}$	$\{Int, \top\}$
4	4	`Invoke v, i, LtOrEq, 100`	$\{Int, \top\}$	$\{Int, \top\}$
5	5	`JumpIf v, label1`		$\{Int, Bool\}$
6	6	`Return void`		$\{Int, Bool\}$

At step 2, before the `Jump`, T_c has been inferred as $[Int, \top]$. At step 3, the program pointer is included in the label descriptors thus, T_c is overloaded by the external table T_{extern}. At step 4, the transfer function associated with the method `LtOrEq` applied on an *Int* changes the type of the return parameter to a *Bool*. At step 5, the precondition to a `JumpIf` is to have a *Bool* value as parameter, which is satisfied.

The complexity of the verification process is $\theta(n)$ due to the fact that the algorithm verifies sequentially the instruction. Furthermore, the memory usage is reduced because the algorithm does not need to store the variable types for every *pp*. It only needs the variables types of the current *pp*. If the type information and the applet have been maliciously modified, the verifier always refuses the code. In order to prove such an assertion we provide a model of our verifier using a formal method (see next Section). The aim of the model is to guarantee a correct implementation of the verifier. It is clear that the verification process is only valid for a given dynamic semantics. Rigorously, we have to model the dynamic execution of the FACADE code, and ensure the coherence between the static and the dynamic semantics. This remains our important topic for further work.

3.2 Modeling the Verifier

As said previously, the only weak point is the implementation of the verifier. We use a formal method in order to gain confidence in our implementation. Currently, the work is on progress and we do not have a proved verifier. We expect to complete our model soon.

The B Method The B Method is a formal method for software engineering developed by J. R. Abrial [1]. It is a *model oriented* approach to software construction. This method is based on the set theory and the first order logic. The basic concept is the *abstract machine* which is used to encapsulate the data describing the state of the system. Invariants can be expressed on the state of the machine which can only be accessed by the specified operations.

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

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

Assumptions and Properties The main idea is to give an abstract specification of the verification process with some key invariants and to refine the model in order to obtain a proved executable program that satisfy the specification and their invariants. In fact, we have to translate the informal specification of the verifier into a more rigorous form which is very close to the work described in [4]. This first step helped us to find some ambiguities in the specification.

We have to prove that our model satisfies the following properties: (i) the algorithm terminates, and either each instruction has been successfully verified or an error has been detected, and (ii) the type inference is coherent with the label at each synchronization point or an error is detected.

The B Model Due to the lack of space we cannot provide the complete B machines of our model. We give hereafter only the different refinements of the operation *bcv_next* linked with the verification of the `Invoke` instruction. At the more abstract level (see B Model 1), this operation states that the error flag may be set and the program counter *pp* may be modified. Such operation can only be activated if the end of the method is not reached.

B model 1. Abstract level of the `Invoke` verification operation: *bcv_next*

$$
\begin{array}{l}
bcv\_next \;= \\
\quad \textbf{SELECT} \quad (method\,(pp) = Invoke) \wedge (End = \textbf{FALSE}) \\
\quad \textbf{THEN} \\
\qquad pp :\in \mathcal{N} \\
\qquad \parallel error \quad :\in \textbf{BOOL} \\
\quad \textbf{END} \;;
\end{array}
$$

At the second level (see B Model 2), we introduce a set: $isVerified \subseteq Nat$, and each time an instruction is checked its *pp* is added to this set. In this refinement we express how the program counter evolves. It specifies only one of the error case: if an instruction is verified a second time.

B model 2. First level of refinement of the bcv_next operation

```
bcv_next =
  SELECT     (method (pp) = Invoke) ∧ (End = FALSE)
   THEN
    IF     (pp ∉ isVerified )
      THEN    isVerified := isVerified ∪ {pp}
      ; error :∈ BOOL
      ; pp := pp + 1
      ELSE    error := TRUE
      END
   END ;
```

The algorithm termination can arise under two conditions (see B Model 3): an error has been detected, or the last instruction has been successfully verified. In the later case, all the pp are elements of the set $IsVerified$ and the last instruction is a $Return$.

B model 3. Terminaison invariant

```
  (Fin = TRUE
      ⇒ ((error = TRUE) ∨ ((method (pp) = Return) ∧ (isVerified =
dom(method)))))
```

At the third level (see B Model 4) starts the verification process by itself. It used several small operations located in an included abstract machine. The operation $ppIsaLabel$ checks if the current pp is referenced as a label. In a such case, the algorithm verifies that the types locally inferred in Tc, are conform with those stored in the T_{extern}. If the type comparison is correct then the local table is replaced by the external table.

B model 4. Second level of refinement of the *bcv_next* operation

```
bcv_next =
  SELECT     (method (pp) = Invoke) ∧ (End = FALSE)
   THEN
    IF     (pp ∉ isVerified)
      THEN     isVerified := isVerified ∪ {pp}
      ; isaLabel ← ppIsaLabel (pp)
      ; IF (isaLabel = TRUE)
          THEN
          TheLabel ← SearchLabel (pp)
          ; error   ← VerifyType (LabelDsc( TheLabel))
          ; IF (error = FALSE)
              THEN  ApplyRuleInvoke (pp)
          END
          ELSE ApplyRuleInvoke (pp)
      END
      ; pp := pp + 1
      ELSE
      error := TRUE
    END
  END ;
```

The *verifyType* operation (see B Model 5) is in the machine that encapsulates the T_c table. The precondition of the operation verifies the type correctness of the argument. Then, all the elements of the local table T_c are compared with the external table using the *SubClass* function. This function verifies the conformity of two types using the class hierarchy.

B model 5. Type comparison procedure

```
typeOk ← verifyType (TExtern) =
  PRE
     TExtern ∈ 1 .. nbTmp → TYPE
  THEN
    ANY LocRes WHERE
      LocRes ∈ BOOL     ∧
      ∀ ii.(ii ∈ dom(Tc )
           ⇒ LocRes = bool( Tc (ii) ∈ SubClass (TExtern (ii))))
    THEN
      typeOk := LocRes
    END
  END ;
```

As explain previously our work on the formal model is in progress. Currently, the system has generated 95 non obvious proof obligations and, all of them have been proved using the theorem prover.

4 Conclusions and Future Works

The FACADE project is an academic research effort supported by the Gemplus Research Lab. It investigates the issue of software production in the field of smart cards. We have introduced the specific domain of smart cards. We have reviewed the programming techniques used for the production of traditional cards and open smart cards. Then, we have presented the FACADE framework for producing smart cards programs.

This framework is based on a typed intermediate language dedicated to smart cards. The first advantage of this single intermediate format is that it enables migration paths between open and traditional cards, from a variety of source languages towards a variety of smart card runtime environments and hardware platforms. There are also other advantages in using a typed intermediate language. First, a rigorous type system can be used to verify the safety of a program. Second, it is possible to abstract the programmers from a single source language if programs of different surface languages share the same runtime system based on a uniform type system. Finally, type safe languages have been shown to support fully optimizing code generation and can efficiently implement security extensions (access control lists or capabilities).

This paper focused on the type-safety verification process. We have shown that it is split in two parts. The off-card part is resource consuming but it generates important information (the labels) in order to minimize the second part of the verification process. By this mean, the on-card part is able to perform the verification with the reduced features of a smart card. We have provided a formal description of this process and we have partly modeled it with the B language. For the moment, we are still working on the complete B model of the verifier. The expected result is a proof of the correctness of this verification process.

Future works are experimental researches on the execution environment of FACADE programs. They include the generation of target code from the FACADE language and the implementation of the runtime system libraries. This work will give us metrics on code size and efficiency, but also an implementation of the operational semantics. In order to prove also the correctness of the execution process, we intend to develop a formal model of the dynamic semantics.

Acknowledgments

We first thank Éric Vétillard for providing us with material to write some parts of this paper, and also Patrick Biget for his helpful comments on this paper. But, Éric must undoubtedly be acknowledged for his careful reading of the paper and his insightful comments which helped us to improve the paper greatly.

References

1. Abrial, J. R. *The B Book. Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. Dwyer, M. *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs.* PhD thesis, University of Massachusetts, Sept. 1995. http://www.cis.ksu.edu/~dwyer/papers/thesis.ps.
3. International Organization for Standardization. *International Standard ISO/IEC 7816: Integrated circuit(s) cards with contacts, parts 1 to 9*, 1987-1998.
4. Lanet, J.-L., and Requet, A. Formal Proof of Smart Card Applets Correctness. In Quisquater and Schneier [10]. http://www.gemplus.fr/developers/passwd_protected/trends/publications/-formproof/art3.htm.
5. Lindhom, T., and Yellin, F. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Sept. 1996.
6. Maosco Ltd. *"MultOS" Web site.* http://www.multos.com/.
7. Microsoft Corporation. *"Smart Card for Windows" Web site.* http://www.microsoft.com/windowsce/smartcard/.
8. Morrisett, G., Walker, D., Crary, K., and Glew, N. From System F to Typed Assembly Language. In *25th Symposium on Principles of Programming Languages* (San Diego, CA, USA, Jan. 1998). http://simon.cs.cornell.edu/Info/People/jgm/papers/tal.ps.
9. Necula, G. C. Proof-Carrying Code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France, Jan. 1997). http://www.cs.cmu.edu/~necula/popl97.ps.gz.
10. Quisquater, J.-J., and Schneier, B., Eds. *Third Smart Card Research and Advanced Application Conference* (Louvain-la-Neuve, Belgium, Sept. 1998). Preproceedings edition.
11. Rose, E. Towards Secure Bytecode Verification on a Java Card. Master's thesis, University of Copenhagen, Sept. 1992. http://www.ens-lyon.fr/~evarose/speciale.ps.gz.
12. Rose, E., and Rose, K. H. Lightweight Bytecode Verification. In *Formal Underpinnings of Java, OOPSLA'98 Workshop* (Vancouver, Canada, Oct. 1998). http://www-dse.doc.ic.ac.uk/~sue/oopsla/rose.f.ps.
13. Shao, Z. Typed Common Intermediate Format. In *USENIX Conference on Domain-Specific Languages* (Barbara, CA, USA, Oct. 1997). http://flint.cs.yale.edu/flint/publications/tcif.html.
14. Sun Microsystems, Inc. *Java Card 2.0 Language Subset and Virtual Machine Specification, Programming Concepts, and Application Programming Interfaces*, Oct. 1997. http://java.sun.com/products/javacard/.
15. Sun Microsystems, Inc. *Java Card 2.1 Virtual Machine, Runtime Environment, and Application Programming Interface Specifications*, Public Review ed., Feb. 1999. http://java.sun.com/products/javacard/public_review.html.
16. Sun Microsystems, Inc. *Java Card 2.1 Virtual Machine Specification*, Draft 2 ed., Feb. 1999. http://java.sun.com/products/javacard/JCVMSpec.pdf.