Specification and Correctness of a (small) Smart Card Operating System

Gustavo Betarte, Cristina Cornes, Nora Szasz, and Alvaro Tasistro

Instituto de Computación
Universidad de la República
Montevideo, Uruguay

Abstract. This work reports an experiment on the specification and formal verification of an operating system variant for smart cards.

1 Introduction

Programs embedded in micro-controllers constitute an important class of critical applications. In these applications the code is executed on a processor of limited resources, the programs are small, critical, and they interact with an external agent. The computational capacity of that agent can vary according to the application in question. Depending on the application the agent can be considered inoffensive or a potential attacker. Implantable biomedical devices, like pacemakers, and smart cards, like phone or purchasing cards, are respectively examples of those applications.

The security problem of smart cards can be addressed at several levels: correctness of the hardware, security of the communication protocol and encryption algorithms, and correctness of the implementation of the smart card embedded operating system. Several works have been done in the former two domains. Some examples of hardware verification have been reported in [Moo92,BWAH90]. Experiences in the study of communication protocols and encryption algorithms can be found in [Bol97a,Bol97b]. This article reports an experiment on the specification and formal verification of a variant of a smart card operating system.

The most significant results achieved are the following:

- We have developed a formal specification of the operating system. It is written in a mathematical language based on Constructive Type Theory [ML84,Coq85]. We shall call this specification the *mathematical specification.*
- We have used the proof-assistant Coq [Bar97] to mechanically verify the mathematical specification as well as the proofs of some interesting properties of the operating system.

The rest of the article proceeds as follows: in section 2 we give a concise description of smart cards and their operating systems. In section 3 we present a

detailed account of the specification. In section 4 we prove that the specification satisfies a security property of the operating system. Finally, we conclude and discuss possible further work.

2 Smart Cards

2.1 History

Plastic cards became popular in the beginning of the 50's. Originally, the functionality of these cards was quite simple: they were mainly used as data containers. The incorporation of a magnetic band on them made it possible to store digital information that could be retrieved by means of reading devices. However, this technology has a serious drawback: the information stored in the card can be read, written or even removed by anyone with access to an adequate device. For this reason, magnetic cards are not the best device in which to store confidential data. Extra precautions are needed to ensure confidentiality. It is for this reason that most systems that use magnetic cards are connected on-line to a host computer on which most of the critical information resides. This, in turn, generates a cost related with information transmission.

A smart card is, in simple terms, a card with an embedded chip. This kind of cards became the ideal device to provide a high level of security based on cryptography. It is possible to store secret keys in them and to execute cryptographic algorithms on those keys. In addition, the possibility of reprogramming a card already on service to incorporate new functionalities allows to think of a range of applications that go beyond the traditional magnetic cards.

2.2 The operating system of a smart card

The operating system of a smart card has as first goal the safe execution of programs and the protection of the information stored in the card. In contrast with general purpose operating systems, these systems provide neither user interface nor the possibility of retrieving externally stored information. The programs are written as ROM code. This restricts the programming methodology and, in addition, does not allow to introduce modifications to the processor once the card has been masked. Therefore, the correction of programming errors becomes rather expensive. Then, a considerable amount of time must be invested in debugging and certifying the code quality. The operating system must be at the same time extremely reliable and robust.

An operating system for smart cards performs the following tasks:

- data transmission to and from the card
- manipulation of internal data
- control of instruction execution
- execution of cryptographic algorithms

The operation mode of a smart card is basically an interactive request-answer process with the user, which is usually a terminal. The terminal sends a request (instruction) to the card, the card processes it, produces a result and then sends it back to the terminal as the answer.
The persistent data of the card is stored in EPROM. The file system has a tree structure with directories and files. Each object of the file system has a header, which in turn contains information about the access permission and a body.
All the instructions concerned with file manipulation make reference to a current file, the *selected file*. The basic operations are those of selection, reading, writing and modification of files.
The memory space in a smart card is so restricted that usually, in contrast with regular operating systems, not all the instructions and file structures can be implemented.
It is for this reason that different profiles have been introduced for the most relevant standards for operating systems.
In addition to the file manipulation instructions, there are, among others, instructions for identification, authentication, execution of cryptographic algorithms and even special instructions for the programming of smart cards or specific applications.
The security is implemented by means of request-answer protocols and a personal identification number (PIN) associated to the card.
The operating system has an automaton that describes all the card's possible states, and for each state the instructions and access allowed.
The card can reach a *blocked* state. In this case no information can be communicated from the card to the exterior. This may happen, for instance, if the maximum number of PIN verification failures is reached.

3 Formal Specification of a Smart Card operating System

In this section we present the specification of a variant of a smart card operating system. This specification aims at clarifying what a card is as a computing device. We used as a basis a reference manual on the state of the art of smart cards [RE97]. This book contains a general description of smart card operating systems. This description is very imprecise and is not enough to specify or implement any of the operating system variants presented in it.
For our specification we chose a variant of the profile P, introduced in the operating systems standard ISO/IEC 7816-4. This profile contains the following instructions: READ BINARY, UPDATE BINARY, SELECT FILE using AID, VERIFY and INTERNAL AUTHENTICATE.

We specify cards of profile P, minus the instruction INTERNAL AUTHENTICATE plus an instruction SELECT CHILD. This specification is for cards of unique application, this latter modelled by a state automaton.
A first sketch follows:

A card comes with a number of *constants*. These are:

- the structure of directories,
- the PIN,
- the maximum number of consecutive PIN verification failures.

The card has a *memory* formed by:

- the data in the files,
- the currently selected file,
- the counter of consecutive PIN verification failures.

The memory is the updatable data of the card. The *instructions* work on the memory using the card constants. These affect the state of the memory and produce *responses*.

The card also comes with a *state automaton*. For each state of the automaton the following are defined:

- a set of permitted instructions,
- for each permitted instruction, a pair of successor states: one in case of successful execution of the instruction and the other for the case of its failed execution.

In the rest of the section we present the formal specification of the operating system. First we discuss the language used to produce the specification. Section 3.2 is devoted to the constants and memory of the card. We start by specifying what a directory structure is. Given this as a basis, we state in a straightforward manner what the constants and the memory are. We define a number of relations on states of the memory that are needed for the specification of the instructions. The instructions are presented in section 3.3. Their semantics is specified in terms of pre- and post-conditions. In section 3.4 we specify what a card automaton is and then define what it is to execute each individual instruction given states of the automaton and the memory. Finally, in section 3.5, the specification of the workings of the whole card is that of a function receiving

- a state of the automaton,
- a state of the memory and
- a stream of instructions or reset messages

and producing a stream of memories and responses resulting from the sequential execution of the input instructions.

3.1 The Language Used

Everything written below can be formalized in Constructive Type Theory [ML84,Coq85]. We have spent some effort to make the specification immediately accessible to anyone used in classical set theory. This should work if only *types* and *sets*, as used in the text, are uniformly interpreted as sets of the classical theory. There remain, however, some (hopefully minor) mismatchings, on which we now proceed to comment.

- We sometimes use the symbol : instead of $\in$. The distinction can be regarded as totally immaterial.
- We use record types, just as in ordinary programming languages, as tuple sets. The selection of a field F out of a record r will be written F_r.
- When specifying functions or records, we do not force ourselves to give names to every set involved. We usually write $f{:}(x{:}A \mid P(x)) \to \langle y{:}B \mid Q(x,y)\rangle$, which should be read: *f maps elements x of set A that verify P to objects y of type B such that $Q(x,y)$.*
- We use some predefined type (set) constructors. In general, these are unproblematic given that we use common notation of programming languages. One possible exception is that of the types of sequences (lists). We write $[A]$ for the type of lists (of arbitrary finite length) whose elements are of type A. If the lists are all of a given length n then we write the corresponding type $[A]_n$. And, finally, if the lists are infinite (streams) we write $[A]_\infty$.
- At one point we use a type called Prop of whom it is evident that must have *propositions* as objects. The classical set theoretic minded reader can take this as just an abuse of notation (as it were a escape into meta-language).

A *specification* of the kind that we are about to examine is ultimately a set (type). It is satisfied (implemented) by any of its elements. In our case, the objects to be specified (cards) have a number of components, related in certain ways. We shall describe this structure, giving specifications of the components and of the relations they must satisfy.

We expect that no further clarification of our language will be necessary. Here are, however, some supplementary comments. Some of the comments are directed to the type theoretic minded reader. These are written in italics.

- In type theory, a sharp distinction is made between types and objects. This is particularly noticeable when considering functions. Functions are not sets but, rather, they are programs in the very concrete sense of functional programming. Moreover, they are in all cases terminating, i.e. there are no partial functions.
- We use one and the same symbol (the ordinary $=$) to denote equality of elements of any type. In particular in the case of functions, we mean their extensional equality. *The extensional equalities on function types that we need are unproblematic to define.*
- *At some points we rely on the validity of excluded middle for some propositions. This is perfectly right since, in all such cases, the predicates involved can be shown to be decidable.*

3.2 Constants and Memory

Directory structures Given a binary relation R on set A we write $R[a]$ for the image of $a \in A$ under R. We use the same notation when the objects involved are subsets of A instead of elements, or functions instead of relations.

Let Aid be the type of application identifiers (AIDs) and Fid that of file identifiers (FIDs). A *directory structure (DS)* consists of:

- A set $\mathcal{A}$ (of valid addresses of files),
- $mf \in \mathcal{A}$ (the root or master file),
- a partition of $\mathcal{A}$ into three sets MF, DF and EF (the file types) such that MF contains mf as its only member,
- an injective function $\mathsf{AD} : DF \to Aid$ (the unique AIDs of DFs),
- a function $\mathsf{AF} : \mathcal{A} \to Fid$ (the file names) and
- a binary relation F on $\mathcal{A}$ (the structure of directories)

such that: For any $a \in \mathcal{A}$ and $b_1, b_2 \in \mathsf{F}[a]$ if $\mathsf{AF}(b_1)=\mathsf{AF}(b_2)$ then $b_1=b_2$ *(Unicity of FID among brothers).*
Given a directory structure, define $\overline{Aid} \stackrel{\text{def}}{=} \mathsf{AD}[DF]$. Then there is DA, the inverse of AD, defined on $\overline{Aid}$ into $\mathcal{A}$.

Let us also define $\mathsf{Ch} \subseteq Fid \times \mathcal{A}$ such that $f\ \mathsf{Ch}\ a$ holds whenever f is the name of a child of a. In symbols: $f\ \mathsf{Ch}\ a \stackrel{\text{def}}{=} f \in \mathsf{AF}[\mathsf{F}[a]]$.

Then for any $a \in \mathcal{A}$ and f:Fid such that $f\ \mathsf{Ch}\ a$ there is a unique address b whose name is f. Let us call FA the function we have just described. In symbols:

$$\mathsf{FA} : (a \in \mathcal{A}, f{:}Fid \mid f\ \mathsf{Ch}\ a) \to \langle b \in \mathcal{A} \mid a\mathsf{F}b \wedge \mathsf{AF}(b) = f \rangle$$

Finally, we allow ourselves in the sequel to use MF, DF and EF either as sets or as predicates on $\mathcal{A}$, according to convenience.

Constants of the card Let PIN be the type of personal identification numbers (PINs, often sequences of four decimal digits). The *constants* of the card are:

- ds:DS (the directory structure),
- $thePin$:PIN (the PIN of the card) and
- $maxEC$:N (the maximum PIN verification failures allowed).

Memory of the card The memory of the card has as first component the file data. The type of file data is as follows:

$FD \stackrel{\text{def}}{=} \mathcal{A} \to \langle len{:}N;\ info{:}[Byte]_{len} \rangle$ i.e. we have length and contents of each file.

We define now the type of the *memory* of the card:

$\mathcal{M} \stackrel{\text{def}}{=} \langle fd{:}FD\,;\ sel{:}\mathcal{A}\,;\ ec{:}[0..maxEC] \rangle$

We refer in the text to the preceding components as follows:

- fd is the *(card) file data*,
- sel is *(the address of) the selected file*,
- ec is the counter of consecutive PIN verification failures or the *error counter*.

We define equivalences between memories that will be useful for specifying the instructions of the card. These equivalences are just equality of memories up to each of its components. More precisely, we introduce:

- $\sim_{sd}$, such that $m \sim_{sd} m'$ holds whenever m and m' differ at most in the contents of their (common) selected file. In symbols:
  $m \sim_{sd} m' \stackrel{\text{def}}{=} (\forall a : \mathcal{A})((a \neq sel_m \supset fd_m(a) = fd_{m'}(a))$
  $\wedge\ sel_m = sel_{m'} \wedge ec_m = ec_{m'}$
- $\sim_{sel}$, such that $m \sim_{sel} m'$ holds whenever m and m' differ at most in (the addresses of) their respective selected files.
- $\sim_{ec}$, such that $m \sim_{ec} m'$ holds whenever m and m' differ at most in the value of their respective error counters.

We omit the formal definition of the last two relations above, given that they are just straightforward.

Finally, we define what it is for a card to be blocked. This is a predicate on the memory defined simply as follows:

$Blocked(m) \stackrel{\text{def}}{=} (ec_m = maxEC)$.

3.3 Instructions

The syntax of instructions is given by the set Ins, defined inductively by the following constructors:

- $SEL_{AID} : Aid \to Ins$
- $SEL_{CHILD} : Fid \to Ins$
- $READ_{BIN} : (o, n{:}N) \to Ins$
- $UPD_{BIN} : (o, n{:}N, upd{:}[Byte]_n) \to Ins$
- $VERIFY : PIN \to Ins$

The semantics of the instructions is specified by pre- and postconditions. For each instruction, the precondition will be a predicate on the memory and the postcondition a relation involving the states of the memory before and after execution, as well as the data extracted from the card. This data will be always a sequence of bytes. We define:
$data \stackrel{\text{def}}{=} [Byte]$

More precisely now, we will have families of propositions:

$\mathcal{P} : (i{:}Ins, m{:}\mathcal{M}) \to \mathsf{Prop}$ and
$\mathcal{Q} : (i{:}Ins, m, m'{:}\mathcal{M}, d{:}data) \to \mathsf{Prop}$.

The instances of these families corresponding to an instruction i will be written $\mathcal{P}_i$ and $\mathcal{Q}_i$ and will be respectively the pre- and postcondition of i. This means, more precisely, that the card must come with an implementation of instructions $\| \_ \|$ that for any instruction i gives a function specified as follows:

$$\| i \| : (m{:}\mathcal{M} \mid \mathcal{P}_i(m)) \to \langle m'{:}\mathcal{M}\ ; d{:}data \mid \mathcal{Q}_i(m, m', d)\rangle$$

For any instruction i and memory m the proposition $\mathcal{P}_i(m)$ is defined by

$$\mathcal{P}_i(m) \stackrel{\text{def}}{=} \neg Blocked(m) \wedge \widehat{\mathcal{P}}_i(m)$$

where $\widehat{\mathcal{P}}$ is an auxiliary family of propositions of the same type as $\mathcal{P}$.

The following table defines the families $\widehat{\mathcal{P}}$ and $\mathcal{Q}$:

i	$\widehat{\mathcal{P}}_i(m)$	$\mathcal{Q}_i(m, m', d)$
$SEL_{AID}(aid)$	$aid \in \overline{Aid}$	$m \sim_{sel} m'$ $\wedge\ sel_{m'} = \mathsf{DA}(aid)$ $\wedge\ d = [\,]$
$SEL_{CHILD}(fid)$	$fid\ \mathsf{Ch}\ sel_m$	$m \sim_{sel} m'$ $\wedge\ sel_{m'} = \mathsf{FA}(sel_m, fid)$ $\wedge\ d = [\,]$
$READ_{BIN}(n, o)$	$EF(sel_m)$ $\wedge\ o{+}n \leq len_{fd_m(sel_m)}$	$m = m'$ $\wedge\ d = info_{fd_m(sel_m)}{\downarrow_{o,n}}$
$UPD_{BIN}(o, n, upd)$	$EF(sel_m)$ $\wedge\ o{+}n \leq len_{fd_m(sel_m)}$	$m \sim_{sd} m'$ $\wedge\ len_{fd_m(sel_m)} = len_{fd_{m'}(sel_{m'})}$ $\wedge\ info_{fd_m(sel_m)} \hookrightarrow_{o,n,upd} info_{fd_{m'}(sel_{m'})}$ $\wedge\ d = [\,]$
$VERIFY(p)$		$m \sim_{ec} m'$ $\wedge\ (p = thePin \supset ec_{m'} = 0$ $\quad \wedge\ p \neq thePin \supset ec_{m'} = ec_m + 1)$ $\wedge\ d = [\,]$

The first two entries are understood straightforwardly. It is only necessary to recall the definitions given when introducing directory structures.
In the third entry we use the function $\downarrow$ which applies to a sequence and parameters o and n. It projects the sequence into the subsequence of length n at offset o. It is just a matter of routine to give a formal description of this function and hence we prefer to omit it. This function has as a particular case the one that selects the element of the given sequence at offset o, i.e. with $n = 1$. We shall denote this particular case in the same way as the general one, only that omitting the second parameter.
Using the $\downarrow$ functions we can define the relation $\hookrightarrow$ used to give the postcondition of the UPDATE BINARY instructions. Let f and f' be sequences of the same length m. Let further o and n be such that $o{+}n \leq m$ and upd a sequence of length

n. Then $f \hookrightarrow_{o,n,upd} f'$ holds whenever f' coincides with f everywhere except at the subsequence of length n at offset o, which must be equal to upd. In symbols: $f \hookrightarrow_{o,n,upd} f' \stackrel{\text{def}}{=} (f'{\downarrow_{o,n}} = upd) \wedge (\forall j \in [0..m-1])(j < o \vee \ j \geq o+n) \supset f'{\downarrow_j} = f{\downarrow_j}$.

Finally, the last entry ensures that the error counter is set to 0 after each successful PIN verification and incremented otherwise.

3.4 Execution of Instructions

Automata An *automaton* $\mathcal{A}$ is determined by:

- S (the set of states),
- s_0 (the initial state),
- $Valid(s) \subseteq \mathcal{I}$ for each $s \in S$ (the set of allowed instructions at each state),
- $\mathcal{T} : (s \in S, i \in Valid(S)) \rightarrow \langle ok,\ fail : S \rangle$, the table of transitions.

The table of transitions specifies two successor states. The first one corresponds to the case of successful execution of the instruction and the other one to that of failed execution.

One Step Execution We now proceed to specify what the *execution of an instruction* is. This notion shall be defined as a function ξ that given a state s, the memory m of the card and the instruction i to be executed, returns a new state s' and a tuple composed by the (possibly) modified memory and the corresponding answer, which is an object of type $\mathcal{R}$. This latter type is defined as follows:

$\mathcal{R} \stackrel{\text{def}}{=} \langle rc{:}RC, data{:}[Byte] \rangle$,

where RC is a set of (return) codes and $[Byte]$ is the type of sequences (or lists) of elements of type $Byte$.
The set of return codes could be chosen in different ways. A a simple alternative is just $RC \stackrel{\text{def}}{=} \{\mathsf{no\ file}, \mathsf{not\ EF}, \mathsf{boundary}, \mathsf{invalid\ instruction}, \mathsf{ack}\}$.

For each instruction i, we specify a table that associates return codes to error conditions on the instruction. This table is referred to below as a relation $ErrMsg_i$ between states of the memory and return codes. It holds of memory m and return code rc whenever an error condition on i (i.e. one that negates the precondition $\mathcal{P}_i$) holds at m and rc is a candidate return code for the case in question. Here is one such table:

i	*Error Condition*	*Return Code*
$SEL_{AID}(aid)$	$aid \notin \overline{Aid}$	no file
$SEL_{CHILD}(fid)$	$\neg(fid\ \mathsf{Ch}\ sel_m)$	no file
$READ_{BIN}(n,o)$	$\neg EF(sel_m)$	not EF
	$o{+}n > len_{fd_m(sel_m)}$	boundary
$UPD_{BIN}(o,n,upd)$	$\neg EF(sel_m)$	not EF
	$o{+}n > len_{fd_m(sel_m)}$	boundary

The two return codes invalid instruction and ack will be employed later.
We specify the function ξ as follows:

$$\xi : (s \in S, m{:}\mathcal{M}, i \in \mathcal{I}) \to \langle s' \in S; out{:}\langle m'{:}\mathcal{M}; \mathrm{r}{:}\mathcal{R}\rangle \mid \chi(s, m, i, s', m', r)\rangle$$

This is the definition of the predicate χ which describes the behaviour of ξ:

$$\begin{array}{rl}
\chi(s, m, i, s', m', r) & \stackrel{\text{def}}{=} \\
\quad i \in \mathit{Valid}(s) \supset & \mathcal{P}_i(m) \supset \ (\langle m', r\rangle = \| \ i \ \| m \wedge \ s' = \mathit{ok}_{\mathcal{T}(s,i)}) \\
& \wedge \ \neg\mathcal{P}_i(m) \supset \ (m' = m \wedge \ \mathit{ErrMsg}_i(m, \mathrm{rc}_r) \\
& \qquad\qquad \wedge \ s' = \mathit{fail}_{\mathcal{T}(s,i)} \wedge \ \mathit{data}_r = [\,]) \\
\wedge \ i \notin \mathit{Valid}(s) \supset & (m' = m \wedge \ r = \mathsf{invalid\ instruction} \wedge \ s' = s \wedge \ \mathit{data}_r = [\,])
\end{array}$$

That is: it has to be checked first whether the instruction is allowed at the current state of the automaton. If this is not the case, a corresponding return code is produced and the sates of the memory and the automaton do not change. If the instruction is allowed, then the result depends on whether its precondition holds or not. If it doesn't, then we have a failed execution: a corresponding return code must be produced and the transition is made to the corresponding successor state of the automaton. If the precondition holds, then the instruction is executed and also the corresponding transition is effected.

3.5 A Card's Life

Finally, we provide the specification of a process *life*, which is intended to represent the potentially infinite working period of the card. The input of this process, from now on called *In*, consists of a state s, a memory m and a stream of elements of the set *InpInst*. This latter type, in turn, is defined as the disjoint union of the set *Ins* and a special instruction that we shall call reset. Observe that reset *is not* an instruction of the profile. Rather, it was introduced to formally reflect the event that occurs when the card is activated by the terminal. The output of the process, that we shall call *Out*, is a stream of tuples, each consisting of a memory and a response. It should be understood as the (infinite) trace resulting from executing the stream of instructions on certain state and memory. How the process acts on the input, and therefore how this latter is related to the output is described by a (coinductive) relation that shall be denoted by $\asymp$.

We start by providing the definition of the types *In* and *Out*:

$$\begin{array}{l}
\mathit{InpInst} \stackrel{\text{def}}{=} [\mathit{Ins} + \{\mathsf{reset}\}]_\infty \\
\mathit{In} \stackrel{\text{def}}{=} \langle s \in S; m{:}\mathcal{M}; is \in \mathit{InpInst}\rangle \\
\mathit{Out} \stackrel{\text{def}}{=} [\langle m{:}\mathcal{M}; r{:}\mathcal{R}\rangle]_\infty
\end{array}$$

Now we specify the process *life* by means of the following declaration:

$$\mathit{life} : (\mathit{in}{:}\mathit{In}) \to \langle \mathit{out}{:}\mathit{Out} \mid \mathit{in} \asymp \mathit{out}\rangle$$

The relation $\asymp$, in turn, is coinductively defined as follows:

$\langle s, m, (\mathsf{reset} :: iss)\rangle \asymp (\langle m', \langle \mathsf{ack}, [\,]\rangle\rangle :: os)$
$\quad \supset m \sim_{sel} m' \wedge sel_{m'} = mf \wedge \langle \mathsf{s}_0, m', iss\rangle \asymp os$

$\langle s, m, (i :: iss)\rangle \asymp (\langle m', r\rangle :: os)$
$\quad \supset \xi(s, m, i) = \langle s', \langle m', r\rangle\rangle \wedge \langle s', m', iss\rangle \asymp os$

4 A property

4.1 An invariant of one step execution

The following proposition shows that any interpreter ξ satisfying the specification, when applied to a blocked memory yields a response containing no data and a blocked memory.

For conciseness we abbreviate by *Correct_Interp* the type

$$(s{:}S)(m{:}\mathcal{M})(i{:}Ins)\ \langle o{:}ResExec \mid \chi(s, m, i, s'_o, m'_o, r'_o)\rangle$$

where *ResExec* denotes $\langle s' \in S; out{:}\langle m'{:}\mathcal{M}; r' : \mathcal{R}\rangle\rangle$

Lemma Consider ξ an interpreter of type *Correct_Interp*, $m{:}\mathcal{M}$ a memory such that *Blocked*(m), $s{:}S$ a state and $i{:}Ins$ an instruction.
Then $data_{r'out_{\xi(s,m,i)}} = [\,]$ and *Blocked*$(m'out_{\xi(s,m,i)})$.

Proof.
The function ξ is of type *Correct_Interp*, then $\xi(s, m, i)$ is a record of type *ResExec* and the following proposition is satisfied

$$\chi(s, m, i, s'_{\xi(s,m,i)}, m'out_{\xi(s,m,i)}, r'out_{\xi(s,m,i)})$$

Consider the abbreviations:

- $s_o \stackrel{\text{def}}{=} s'_{\xi(s,m,i)}$
- $m_o \stackrel{\text{def}}{=} m'out_{\xi(s,m,i)}$
- $r_o \stackrel{\text{def}}{=} r'out_{\xi(s,m,i)}$

Unfolding the definition of χ in $\chi(s, m, i, s_o, m_o, r_o)$ yields the following conjunction (call it $\mathcal{S}$) that must hold.

$$\begin{array}{l} i \in Valid(s) \supset \mathcal{P}_i(m) \supset \langle m_o, r_o\rangle = \| \ i \ \|(m) \wedge s_o = ok_{\mathcal{T}(s,i)} \\ \qquad\qquad \wedge \neg\mathcal{P}_i(m) \supset m_o = m \wedge ErrMsg(i, m, rc_{r_o}) \wedge s_o = fail_{\mathcal{T}(s,i)} \wedge data_{r_o} = [\,] \\ \wedge\ i \notin Valid(s) \supset m_o = m \wedge r_o = \mathsf{invalid\ instruction} \wedge s_o = s \wedge data_{r_o} = [\,] \end{array}$$

We are going to prove that $\mathcal{S}$ implies that $data_{r_o} = [\,]$ and *Blocked*(m_o).
We know that *Valid* is a decidable predicate. The proof proceeds by case analysis on $i \in Valid(s)$.

- Assume $i \in Valid(s)$. Then by the left part of the conjunction $\mathcal{S}$ the following proposition (call it $\mathcal{S}$') holds:

$$\mathcal{P}_i(m) \supset \langle m_o, r_o \rangle = \| \; i \; \|(m) \wedge s_o = ok_{\mathcal{T}(s,\,i)}$$
$$\wedge \neg\mathcal{P}_i(m) \supset m_o = m \wedge ErrMsg(i, m, rc_{r_o}) \wedge s_o = fail_{\mathcal{T}(s,\,i)} \wedge data_{r_o} = [\,]$$

We proceed by case analysis on the proposition $\mathcal{P}_i(m)$.
  - Assume $\mathcal{P}_i(m)$ holds. Then unfolding the definition of $\mathcal{P}_i$ in $\mathcal{P}_i(m)$ yields $\neg Blocked(m) \wedge \widehat{\mathcal{P}}_i(m)$. In particular, $\neg$ $Blocked(m)$ must hold, but this is absurd because by hypothesis we had $Blocked(m)$. Hence, $data_{r_o} = [\,]$ and $Blocked(m_o)$ hold trivially.
  - Assume $\neg\mathcal{P}_i(m)$ holds. Then by the right part of the conjunction $\mathcal{S}$' we obtain directly that $data_{r_o} = [\,]$ and that $m = m_o$. But as $=$ is the identity relation and we know $Blocked(m)$, then $Blocked(m_o)$ holds.

- Assume $i \notin Valid(s)$. Then by the right part of $\mathcal{S}$ we obtain $data_{r_o} = [\,]$ and $m = m_o$. But as $=$ is the identity relation and we know $Blocked(m)$, then $Blocked(m_o)$.

□

4.2 An invariant of the card

The next proposition shows that no information can be extracted out from a blocked card. More precisely it shows that the execution of a blocked card on any stream of input instructions produces a stream of outputs whose responses contain no data.
For the sake of clarity, we rewrite the definition of $\asymp$ in the following equivalent way:

$$\langle s, m, (i :: is)\rangle \asymp (\langle m_o, r_o\rangle ::\text{os}) \;\supset$$
$$(i = \mathsf{reset} \supset m \sim_{sel}\, m_o \wedge sel_{m_o} = mf \wedge r_o = \langle \mathsf{ack}, [\,]\rangle \wedge \langle \mathsf{s}_0, m_o, is\rangle \asymp os)$$
$$\wedge\; (i \neq \mathsf{reset} \supset m_o = m'_{\xi(s,\,m,\,i)} \wedge r_o = \mathrm{r}'_{\xi(s,\,m,\,i)} \wedge \langle s'_{\xi(s,\,m,\,i)}, \mathrm{m}'_{\xi(s,\,m,\,i)}, is\rangle \asymp os)$$

We start with an auxiliary definition. We say that a stream of outputs *"has no data"* if the response contained in each element of the stream is the empty sequence of bytes. The predicate $NoData : Out \to \mathsf{Prop}$ is coinductively defined by:

$$NoData((\langle m, r\rangle :: s)) \supset data_r{=}[\,] \wedge NoData(s)$$

Proposition Let $in = \langle s, m, (i :: is)\rangle$:$In$ be such that $Blocked(m)$ and o:Out such that $in \asymp o$. Then $NoData(o)$.

Proof.
We have to prove that each element of the stream o has a response with no data. By hypothesis we know that $in \asymp o$, therefore by definition of $\asymp$ the object o must be of the form $(\langle m_o, r_o\rangle ::\text{os})$. The proof proceeds in two steps.

1. We prove that $data_{r_o} = [\,]$ and $Blocked(m_o)$. We reason by case analysis on i.
   (a) Assume $i = \mathsf{reset}$. Then, by definition of $\asymp$ we know that
   - r_o must be of the form $\langle \mathsf{ack}, [\,]\rangle$, therefore $data_{r_o}{=}[\,]$.

- $m_o \sim_{sel} m$. By definition of $\sim_{sel}$, $ec_m = ec_{m_o}$. We know $Blocked(m)$ by hypothesis, then necessarily $Blocked(m_o)$ holds.

(b) Assume $i \neq$ reset. Then by definition of $\asymp$ $m_o = m'_{\xi(s,m,i)}$ and $r_o = r'_{\xi(s,m,i)}$. Thus, by the lemma we obtain $Blocked(m'_{\xi(s,m,i)})$ and $data_{r'out_{\xi(s,m,i)}} = []$. Finally, by transitivity we have $Blocked(m_o)$ and $data_{r_o} = []$.

We have proved that $r_o.data = []$ and that $Blocked(m_o)$.

2. To finish the proof of $NoData(o)$ it remains to show that $NoData(os)$. We proceed by case analysis on i. If $i = reset$ (reasoning like in step 1a) this follows from $\langle s_0, m_o, is \rangle \asymp os$ and the fact that m_o is blocked. If $i \neq reset$, (reasoning like in 1b) this follows from $\langle s'_{\xi(s,m,i)}, m'_{\xi(s,m,i)}, is \rangle \asymp os$ and the fact that $m'_{\xi(s,m,i)} = m_o$ is blocked.

□

5 Conclusion

We have produced a completely verified specification of a profile P variant for smart cards. We have also constructed the proof of an important property that must be satisfied by any implementation that fulfills the specification.

The mathematical setting provided by the language used to write down the specification allowed us to formalize the problem in a complete and rigorous manner. The specification coding in terms of the language provided by Coq was straightforwardly written. The environment and tools provided by the editor were quite helpful, specially when constructing the proof. The Coq sources of the specification and proofs are available at `http://www.fing.edu.uy/~mf/SmartCards`.

In [Off95] different levels at which formal methods can be introduced in the software development process are discussed and analyzed. Several works in the literature study the verification of hardware and software components similar to those embedded in smart cards. We are not aware, though, of previous work on the application of formal methods to analyze smart card operating systems. The specification of the operating system of a smart card in a formal language would be useful to detect errors in the informal specification of instructions, to have a formal counterpart to evaluate completeness and soundness of test plans and to improve confidence in the code inspection steps of the operating system implementation.

Regarding further work we are interested in detecting and proving some other properties that must hold for any implementation that satisfies the specification. We identify as a natural continuation of this work the use of the specification obtained to derive and extract certified programs implementing the operating system. In Coq it is possible to automatically extract functional code from the proof that the specification is satisfiable. A more ambitious task would be to write down and verify an imperative language implementation of the system

and prove it correct with respect to the specification. We think this could be done using the tools described in [Fil99].

We have started investigating a particular class of smart cards, those known as *Java Cards* [Car99]. Those cards admit their applications being completely written in the Java language [GJS96], which incorporates a wide range of functionalities to smart card technology. We think that the results here reported could be a good startpoint to formulate a specification of this family of cards.

References

[Bar97] B. Barras et al. The Coq Proof Assistant Reference Manual, Version 6.1. Rapport de recherche 203, INRIA, 1997.

[Bol97a] D. Bolignano. Towards a Mechanization of Cryptographic Protocol Verification. In 9th. International Computer-Aided Verification Conference, June 1997.

[Bol97b] D. Bolignano. Towards the Formal Verification of Electronic Comerce Protocols. In 10th. Computer Security Foundations Workshop, June 1997.

[BWAH90] B. Brock and Jr W. A. Hunt. Report on the Formal Specification and Partial Verification of the VIPER Microprocessor. Technical Report 46, Computational Logic, Inc., January 1990.

[Car99] Java Card. Java Card Technology. `http://java.sun.com/products/javacard/`, 1999.

[Coq85] T. Coquand. *Une Théorie des Constructions*. Thèse de doctorat, Université Paris 7, 1985.

[Fil99] J.C. Filliatre. Proof of Imperative Programs. In *Chapter 18 of* The Coq Proof Assistant Reference Manual, Version 6.2.4, 1999.

[GJS96] J. Gosling, B. Joy, and G. J. Steele. *The Java^{TM} Language Specification*. Addison-Wesley, 1996.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[Moo92] J.S. Moore. Mechanically Verified Hardware Implementing an 8-Bit Parallel IO Byzantine Agreement Processor. Technical Report NASA Contractor Report 189588, Computational Logic, Inc., 1992.

[Off95] Office of Safety and Mission Assurance. Formal Methods Specification and Verification Guidebook for Software and Computer Systems. Volume I: Planning and Technology Insertion. Technical Report NASA-GB-002-95, Release 1.0, NASA, July 1995.

[RE97] W. Rankl and W. Effing. *Smart Card Handbook*. John Wiley & Sons, 1997.