Specification of a Smart Card Operating System

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

Department of Computing Science
Universidad de la República
Montevideo, Uruguay

Abstract. We present a specification in Type Theory of a variant of a standard for smart cards' operating systems. The specification has been completely formalized and a basic property concerning security of the card has been developed and mechanically verified using the proof assistant Coq.

1 Introduction.

Programs embedded in micro-controllers constitute a significant class of applications. In these cases the code is executed on a processor of limited resources and the programs are small and normally critical. They interact with an external agent whose computational capacity can vary according to the application in question. Also depending on the application the agent can be considered inoffensive or a potential attacker. Implantable biomedical devices, like pacemakers, are examples of the first kind, whereas smart cards, like phone or purchasing cards, are examples of the second. The development of these devices requires high investment in quality assurance, since error correction is always very expensive and, in cases, even next to unfeasible. Therefore, there arises a significant interest in investigating formal methods with a view to make them part of construction processes to be established as standards.

In the case of smart cards, security constitutes the central aspect of the problem of correctness. This problem 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 operating system embedded in the card. Several works have been done in the former two domains. Some examples of hardware verification have been reported in [10, 4]. Experiences on the formal treatment of communication protocols and encryption algorithms can be found in [2, 3]. The present article reports on an experiment of specification of a smart card operating system. We are not aware of previous work on the application of formal methods to analyze this kind of systems.

The most significant results achieved are the following:

* {gustun,cornes,nora,tato}@fing.edu.uy

- We wrote a specification of the operating system in a language close to ordinary mathematical language. We call this the *mathematical* specification. It can be understood assuming either ordinary set theory or constructive type theory [9, 5]. The specification possesses a high level of abstraction, since it does not contain formulations of particular algorithms or data structures but rather types of functions and abstract data types.
- We wrote down a complete formalisation of the mathematical specification using the proof assistant Coq [1].
- We developed in Coq (and thus mechanically verified) the proof of a significant basic property of the specified operating system.

The case considered is simple but also quite realistic. After carrying out this work, we believe that the approach employed can be scaled up to larger, more complex examples within acceptable margins of effort. A formal specification like the one we have produced can be used in several ways, all of which constitute interesting possibilities at the present state of the art of the construction of smart cards:

- The specification can be tested through the development and mechanical verification of proofs of required properties.
- It can be used as the foundation for the process of mechanically verifying or deriving actual machine code implementations.
- It can be taken as a standard of specification of smart card operating systems.

The rest of the article proceeds as follows: in section 2 we give a concise account of the state of the art of smart cards and their operating systems. In section 3 we present the mathematical specification. In section 4 we prove that the specification satisfies a security property of the operating system. Finally, we conclude commenting on several aspects of the work carried out and discussing possible further work.

2 Smart Cards.

We proceed now to give a succint informal account of smart cards and their operating systems. Most of this section is based upon [11], which is a reference manual on the state of the art in this subject.

2.1 History

Plastic cards became popular in the beginning of the ‘50s. Originally, the functionality of these cards was quite simple: they were mainly used as data containers. This capacity was enhanced by the incorporation of a magnetic band on the cards, which made it possible to store digital information to be retrieved by means of appropriate 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, e.g. the connection on-line of the system using magnetic cards to a host computer on which most of the critical information resides. This, in turn, generates a cost related to 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 far beyond what is feasible with the traditional magnetic cards.

2.2 The operating system of a smart card

We shall now take a closer look at the structure and operation of smart card operating systems. A first point to be observed is that the process of standardization of these systems finds itself at present at its beginnings, with a number of coexisting international and industry standards. In the reference manual [11], a specific system is used as the basis for the general description of their architecture and functionalities, namely the system called STARCOS, which has been undergoing development since 1990 by the company Giesecke und Devrient and the German Society for Mathematics and Data Processing. Here are the characteristic general features:

- The operation mode of a smart card is basically a request-answer process where the user is a terminal. The terminal sends a request (instruction) to the card, the card processes it, produces a result and sends it back to the terminal as the answer. So, in contrast to general purpose operating systems, smart cards' ones provide neither man-machine interface nor the possibility of accessing external storage media.
- The persistent data of the card is stored in so-called *files* implemented on EEPROM. The basic operations are those of selection, reading and updating of files. All the instructions concerned with file manipulation make reference to a currently *selected* file.
- The file system has a tree structure with directories (called *dedicated* files or *DF* for short) and plain data files (called *elementary files*, or *EF*). The root directory is implicitly selected at the beginning of the operation of the card. It is called the *master* file, or *MF* for short.
- The directories (DFs) normally represent applications, i.e. they group together files or directories that are related to a certain use of the card. The set of applications with which the card is to be equipped is decided at the time of construction or upgrading of the card. In other words, the structure of directories is not to be modified by any user terminal and, therefore, there are no instructions for modifying that structure.

- The directories have identifying names, which are called *application identifiers* or *AIDs* for short. In addition, every file possesses a so-called *file identifier (FID)*. FIDs need not to be unique in the entire file system, but they must be so among the children files of any given directory.
- Every elementary file has a certain structure: it may be of fixed or variable length, linear or cyclic or contain executable code. Each object of the file system has a header, which contains information about properties of the file, e.g. about access permission, and, in the case of elementary files, about its structure. Besides the header there is of course the data of the file. In the case of directories, this data consists just in the identification of the children files of the directory. In general, there are no I/O instructions that operate on directories, i.e. not even reading operations at that level. Therefore, actual I/O occurs only on elementary files.
- 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 memory space in a smart card is so restricted that, in contrast to regular operating systems, it is usually unfeasible to implement all the instructions and (elementary) file structures. It is for this reason that different profiles or variants have been introduced for the most relevant standards for operating systems.
- Security is implemented by means of request-answer protocols and a personal identification number (the *PIN*) associated to the card. Usually, PIN recognition involves execution of cryptographic algorithms. Again, the PIN is only modifiable at masking time, as is the number of consecutive PIN verification failures after which the card becomes blocked. If a card is blocked, no information can be communicated between it and the user terminal.
- It is possible in some cards to restrict the sequences of instructions to be accepted. In other words, it is possible to furnish the card with a number of programs that can be selected and executed. One typical example is given by the process of authentication of a terminal. In this case, it would be desirable that activation of the card leads to a state where only a number capable of constituting a PIN is accepted by the card, which would then proceed to check it. The instructions to be subsequently accepted by the card depend in general of its intented use or, as it is called, application. Therefore, it is said that cards may be single- or multi-application, depending on whether they come with one or more of these programs. The programs are represented as finite automata. Each state of one of these automata specifies the instructions that are allowed, and may as well determine access permissions to files.

3 The Mathematical Specification.

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. The description in [11], which we use as starting point, is very general in the sense of admitting several interpretations (and thus, implementations) of the concepts introduced.

For our specification we chose a variant of the profile called P, introduced in the operating systems standard ISO/IEC 7816-4. It is a simple but nevertheless realistic (i.e. actually existing) case. We have used it as a first test case that allows us to evaluate the feasibility of larger, more complex examples. Another reason to choose it is that these more complex profiles were much less well defined in the source reference manual [11]. In summary, we consider:

- Elementary files of fixed length. More precisely, each elementary file is a sequence of bytes of a fixed length, which depends on the file in question.
- The instructions READ, UPDATE, SELECT using AID, SELECT CHILD and VERIFY. The first two of these are not worth more explanations at this point. The third one allows to select a directory giving its AID. The fourth is for selecting a child of the current file, which must then be a directory. In this case the selection is done via the (unique) FID of the child. Finally, VERIFY comes with a purported PIN, which is to be checked against the card's real one. We have not considered encryption of the PIN, since it does not introduce any interesting problems at this level of our study.
- This specification is for cards of unique application, modelled by a state automaton included in the card. Furthermore, in this case the application has access to the data of any file of the card. Therefore, we can do without considering access permissions.

A first sketch of the specification follows:

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

- the structure of directories,
- the PIN,
- the number of consecutive PIN verification failures after which the card gets blocked. We shall call this the *blocking number*, for short.

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. They affect the state of the memory and produce *responses*.

In addition, we have the automaton representing the application of the card. For each state of the automaton the following are defined:

- a set of permitted instructions and
- 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 a mathematical specification of the operating system. We start by discussing the language used to produce the specification. Section 3.2 is devoted to the constants and memory of the card. We need to define first 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 (finite) 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, we present the specification of the workings of the card as a function receiving

- a state of the automaton,
- a state of the memory and
- a stream of instructions or card activation 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 [9, 5]. In fact, we have accomplished that using the proof assistant Coq. We have spent some effort to make the mathematical 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) mismatches, 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 (labeled) 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 $f{:}(x{:}A \mid P(x)) \rightarrow \langle y{:}B \mid Q(x,y)\rangle$, 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 allow ourselves to use subsets of a given set $\mathcal{A}$ either as sets or as predicates on $\mathcal{A}$, according to convenience.
- 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 some points 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.

The specification to be presented aims at being as abstract as possible while preserving significant expectable properties of the operation of the card. We do this with the purpose of leaving room for a wide range of implementations and achieve conciseness wherever possible. We also aim at not contradicting the informal description of the original text [11], though this is difficult to establish beyond doubt, given the ambiguity inherent to the source. We start with the specification of what a directory structure of a card is. This specification constitutes the basis on which to build that of the actual components of the card.

Directory structures. Given a binary relation $R \subseteq A \times B$ 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, and also for images of such subsets along functions.

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 ("absolute") valid addresses of files),
- a partition of $\mathcal{A}$ into two sets DF and EF (the file types),
- $mf \in DF$ (the root or master file),
- an injective function $\mathsf{AD} : DF \rightarrow Aid$ (the unique AIDs of DFs),
- a function $\mathsf{AF} : \mathcal{A} \rightarrow Fid$ (the file names),
- a binary relation $\mathsf{Ch} \subseteq Fid \times DF$, (the structure of directories) together with
- a function $\mathsf{FA} : (d \in DF, f{:}Fid \mid f \,\mathsf{Ch}\, d) \rightarrow \mathcal{A}$, which gives the (address of the) unique file with FID f that is a child of d.
- a function $\mathsf{len} : EF \rightarrow N$ (the fixed lengths of elementary files).

Instances of Ch will be written using infix notation. Notice that the specification of this relation allows for cycles and hence directory structures are not specified as trees. The reason for this is not any technical difficulty, but just the aim for abstraction and conciseness commented earlier. Also it is not required that directories have children at all.

Given a directory structure, define $\underline{Aid} \stackrel{\mathrm{def}}{=} \mathsf{AD}[DF]$. Then there is DA, the inverse of AD, defined on $\underline{Aid}$ into $\mathcal{A}$.

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 blocking number).

Memory of the card. The memory of the card has as first component the data in the elementary files. The type of this data is specified as follows:

$$FD \stackrel{\mathrm{def}}{=} (a{:}EF) \rightarrow [Byte]_{\mathsf{len}(a)}.$$

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

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

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:

- $\mathsf{selAid} : Aid \rightarrow Ins$
- $\mathsf{selChild} : Fid \rightarrow Ins$
- $\mathsf{read} : (o, l{:}N) \rightarrow Ins$
- $\mathsf{upd} : (o, l{:}N, u{:}[Byte]_l) \rightarrow Ins$
- $\mathsf{verify} : PIN \rightarrow Ins$

Let us give some informal explanations. The two first cases have already been explained. The instruction $\mathsf{read}(o, l)$ reads off the subsequence of length l of the selected file whose starting point is reached by skipping o positions from the beginning of the file (o is called the *offset*). Then the selected file must be an elementary file and at least of the implicitly required length. Similarly, the instruction $\mathsf{upd}(o, l, u)$ replaces the subsequence of the selected file positioned after offset o and having length l by the sequence u.

The semantics of the instructions will be specified by pre- and post-conditions. 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:

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

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. 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 we factor out the condition that the card must be operative for any instruction to be executed and let the pre-conditions particular to each instruction be specified by making use of the auxiliary family $\widehat{\mathcal{P}}$ which is of the same type as $\mathcal{P}$. The following table defines the families $\widehat{\mathcal{P}}$ and $\mathcal{Q}$. Corresponding explanations come next to the table.

i	$\widehat{\mathcal{P}}_i(m)$	$\mathcal{Q}_i(m, m', d)$
$\mathsf{selAid}(aid)$	$aid \in \underline{Aid}$	$m \sim_{\overline{sel}} m'$ $\wedge\ sel_{m'} = \mathsf{DA}(aid)$ $\wedge\ d = [\,]$
$\mathsf{selChild}(fid)$	$fid\ \mathsf{Ch}\ sel_m$	$m \sim_{\overline{sel}} m'$ $\wedge\ sel_{m'} = \mathsf{FA}(sel_m, fid)$ $\wedge\ d = [\,]$
$\mathsf{read}(o, l)$	$EF(sel_m)$ $\wedge\ o+l \leq \mathsf{len}(sel_m)$	$m = m'$ $\wedge\ d = fd_m(sel_m)@(o, l)$
$\mathsf{upd}(o, l, u)$	$EF(sel_m)$ $\wedge\ o+l \leq \mathsf{len}(sel_m)$	$m \sim_{sd}\ m'$ $\wedge\ fd_m(sel_m) \underset{o,l,u}{\longmapsto} fd_{m'}(sel_{m'})$ $\wedge\ d = [\,]$
$\mathsf{verify}(p)$		$m \sim_{ec}\ m'$ $\wedge\ (p = thePin \supset ec_{m'} = 0$ $\wedge\ p \neq thePin \supset ec_{m'} = ec_m + 1)$ $\wedge\ d = [\,]$

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 @ which applies to a sequence and natural numbers o and n. It projects the sequence to its 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 @ functions we can define the relation $\longmapsto$ used to give the postcondition of the update 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 u a sequence of length n. Then $f \underset{o,n,u}{\longmapsto} f'$ holds whenever f' coincides with f everywhere except at the subsequence of length n at offset o, which must be equal to u. In symbols:

$f \underset{o,n,u}{\longmapsto} f' \stackrel{\text{def}}{=} (f'@(o, n) = u) \wedge (\forall j \in [0..m-1])(j < o \vee\ j \geq o + n) \supset f'@j = f@j$.

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

3.4 Execution of Instructions.

We must now specify the execution of instructions as governed by the automaton representing the (single) application of the card. We start by formally introducing automata.

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

- S (the set of states),
- $\mathsf{s}_0 \in S$ (the initial state),
- $\mathit{Valid}(s) \subseteq \mathcal{I}$ for each $s \in S$ (the set of allowed instructions at each state),
- $\mathcal{T} : (s \in S, i \in \mathit{Valid}(S)) \rightarrow \langle ok,\ fail : S\ \rangle$, 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 its 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 (the interpreter) 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. Answers, or as we shall say, *responses* are of the following type:

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

where RC is a set of (return) codes. We need to introduce return codes in order to give answers in cases of failed execution of instructions. The set of return codes could be chosen in different ways. A simple alternative is just:
$RC \stackrel{\text{def}}{=} \{\mathsf{no\ file}, \mathsf{not\ EF}, \mathsf{boundary}, \mathsf{invalid\ instruction}, \mathsf{ack}, \mathsf{nack}\}$.

For each instruction i, we specify a table that associates return codes to error conditions on the instruction. Using this table we define below a relation ErrMsg_i between states of the memory and return codes. This relation 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*
$\mathsf{selAid}(aid)$	$aid \notin \underline{Aid}$	no file
$\mathsf{selChild}(fid)$	$\neg(fid\,\mathsf{Ch}\,sel_m)$	no file
$\mathsf{read}(o,l)$	$\neg EF(sel_m)$	not EF
	$o{+}l{>}\mathsf{len}(sel_m)$	boundary
$\mathsf{upd}(o,l,u)$	$\neg EF(sel_m)$	not EF
	$o{+}l{>}\mathsf{len}(sel_m)$	boundary

To define $ErrMsg_i$ it is enough to state in addition that, for any instruction i, any blocked memory m is associated to the return code nack. The interpreter is specified now as follows:

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

where the predicate χ describes the right relation between the state of the automata and of the memory at activation of the instruction and the corresponding states and output at termination of the execution. Here it is in symbols:

$$\begin{array}{l}
\chi(s,m,i,s',m',r) \stackrel{\mathrm{def}}{=} \\
\qquad i \in Valid(s) \supset \quad (\mathcal{P}_i(m) \supset \ (\mathcal{Q}_i(m,m',d_r) \wedge rc_r = \mathsf{ack} \wedge s' = ok_{\mathcal{T}(s,\,i)}) \\
\qquad\qquad\qquad\qquad \wedge\ \neg\mathcal{P}_i(m) \supset \ (m' = m \wedge ErrMsg_i(m, rc_r) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ s' = fail_{\mathcal{T}(s,\,i)} \wedge d_r = [\,])) \\
\qquad \wedge\ i \notin Valid(s) \supset \quad (m' = m \wedge rc_r = \mathsf{invalid\ instruction} \wedge s' = s \wedge d_r = [\,])
\end{array}$$

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 (bottom line of the formula above) a corresponding return code is produced and the states of the memory and the automaton do not change. Also, no data is produced in this case. If the instruction is allowed, then the result depends on whether its precondition holds or not. If it does, then the postcondition of the instruction must hold of the original state of the memory and the result of the execution, i.e. the new state of the memory and the data produced. The code ack is returned in this case. If the precondition does not hold, then we have a failed execution: a corresponding return code must be produced and no data comes out of the card. In any case in which the instruction is valid for the original state of the automaton, the transition to the corresponding next state 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 consists of a state s, a memory m and a stream of what we shall call *events*. An event is either an instruction or the activation of the card by the terminal, which we shall write reset. The output of the process 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 $\asymp$. Let us now give the formal definitions.

$Events \stackrel{\text{def}}{=} [Ins + \{\mathsf{reset}\}]_\infty$
$In \stackrel{\text{def}}{=} \langle s \in S; m{:}\mathcal{M}; es \in Events\rangle$
$Out \stackrel{\text{def}}{=} [\langle m{:}\mathcal{M}; r{:}\mathcal{R}\rangle]_\infty$

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

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

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

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

$$\langle s, m, (i :: ess)\rangle \asymp (\langle m', r\rangle :: oss)$$
$$\supset (\xi(s, m, i) = \langle s', m', r\rangle \wedge \langle s', m', ess\rangle \asymp oss)$$

Here we write individual record objects directly as tuples, i.e. omitting labels. The first clause stands for the case of an reset event. Then an ack is produced, no data comes out and we enter the initial state of the automaton, i.e. s_0. Also the master file becomes the selected file. The second clause is for the case of an incoming arbitrary instruction, denoted by the variable i. The specification is in this case quite straightforward.

This finishes the specification. Notice that we have not specified the initialization of the card, i.e. a predicate on the memory that must be satisfied when the card is by the first time put to work. This predicate is in our case very simple, i.e. we should only require the error counter to be initialized to 0. Actually, it turns out to be more general and convenient to proceed as we did (i.e. not restricting the initial state of the memory) and introducing further assumptions whenever this is necessary to derive properties in which we are interested.
It now only remains to point out that the specification has been completely formalized using the proof assistant Coq. Comments on this latter process are to be found in the conclusions of the paper.

4 A property.

We wish to end up showing that no data can be extracted out from a blocked card. This is a significant basic property concerning security of the specified cards. Also the proofs that we are about to give have been written, and thereby mechanically verified, using Coq. The presentation below is rather detailed, i.e. much more than what is needed to convince the reader. We present the proofs in that manner in order to show some of the flavour of the completely formalized versions.

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 again a blocked memory and a response containing no data.

We recall that the interpreter of the card has been specified in the following way:

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

Let us write *CorrectInterpreter* for the type of ξ.

Lemma Consider ξ of type *CorrectInterpreter*, m a memory such that $Blocked(m)$, s a state and i an instruction.
Then $d_{r_{\xi(s,\,m,\,i)}} = [\,]$ and $Blocked(m'_{\xi(s,\,m,\,i)})$.

Proof.
The function ξ is of type *CorrectInterpreter*. Then $\xi(s, m, i)$ is a record of the form $\langle s_o, m_o, r_o\rangle$ and we wish to prove $d_{r_o} = []$ and $Blocked(m_o)$.
We have furthermore $\chi(s, m, i, s_o, m_o, r_o)$. Unfolding this we get

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

From the former, we proceed by case analysis on $i \in Valid(s)$ (on the basis that $Valid$(s) is decidable).

- Assume $i \in Valid(s)$. Then we have

  $$\begin{array}{l} \mathcal{P}_i(m) \supset (\mathcal{Q}_i(m, m_o, d_{r_o}) \wedge rc_{r_o} = \mathsf{ack} \wedge s_o = ok_{\mathcal{T}(s,\,i)}) \\ \wedge\ \neg\mathcal{P}_i(m) \supset (m_o = m \wedge ErrMsg(i, m, rc_{r_o}) \wedge s_o = fail_{\mathcal{T}(s,\,i)} \wedge data_{r_o} = [\,]) \end{array}$$

  We proceed by case analysis on the proposition $\mathcal{P}_i(m)$ (which is always decidable).

  - Assume $\mathcal{P}_i(m)$ holds. Then unfolding the definition of $\mathcal{P}_i$ in $\mathcal{P}_i(m)$ yields $\neg Blocked(m) \wedge \widehat{\mathcal{P}}_i(m)$, i.e. in particular, $\neg\ Blocked(m)$. Now this is absurd because we supposed $Blocked(m)$. Hence, $data_{r_o} = [\,]$ and $Blocked(m_o)$ trivially.
  - Assume $\neg\mathcal{P}_i(m)$ holds. Then we obtain directly that $data_{r_o} = [\,]$. On the other hand, we get also $m = m_o$ and, hence, since $Blocked(m)$, it follows $Blocked(m_o)$.

- Assume $i \notin Valid(s)$. Then we again obtain directly $data_{r_o} = [\,]$ and $m = m_o$ and find ourselves again in the latter case.

□

4.2 An invariant of the card's life.

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.

Let us rewrite the definition of $\asymp$ in the following equivalent way:

$$\begin{array}{l}\langle s, m, (e :: es)\rangle \asymp (\langle m_o, r_o\rangle :: os) \ \supset \\ \quad e = \mathsf{reset} \supset (m \sim_{sel} m_o \wedge sel_{m_o} = mf \wedge r_o = \langle \mathsf{ack}, [\,]\rangle \wedge \langle \mathsf{s}_0, m_o, es\rangle \asymp os) \\ \ \wedge\ e \neq \mathsf{reset} \supset (m_o = m'_{\xi(s,m,i)} \wedge r_o = r_{\xi(s,m,i)} \wedge \langle s'_{\xi(s,m,i)}, m'_{\xi(s,m,i)}, es\rangle \asymp os)\end{array}$$

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:

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

Proposition Let $in = \langle s, m, (e :: es)\rangle : In$ 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 assumption we know that $in \asymp o$, therefore by definition of $\asymp$ the object o must be of the form $\langle m_o, r_o\rangle :: os$. The proof proceeds in two steps.

1. We prove that $d_{r_o} = [\,]$ and $Blocked(m_o)$. We reason by case analysis on e.
   (a) Assume $e = \mathsf{reset}$. Then, by definition of $\asymp$ we know that
      - r_o must be of the form $\langle \mathsf{ack}, [\,]\rangle$, therefore $d_{r_o} = [\,]$.
      - $m_o \sim_{sel} m$. By definition of $\sim_{sel}$, $ec_m = ec_{m_o}$. Hence, from $Blocked(m)$ we get $Blocked(m_o)$.

   (b) Assume $e \neq \mathsf{reset}$. Then, by definition of $\asymp$, we obtain $m_o = m'_{\xi(s,m,i)}$ and $r_o = r_{\xi(s,m,i)}$.
      Now, by the lemma, we get $Blocked(m'_{\xi(s,m,i)})$ and $d_{r_{\xi(s,m,i)}} = [\,]$, as we wished to prove.
2. It remains to show that $NoData(os)$. We proceed by case analysis on e.
   - If $e = reset$ (reasoning like in step 1a) this follows from $\langle \mathsf{s}_0, m_o, es\rangle \asymp os$ and the fact that m_o is blocked.
   - If $i \neq reset$, (reasoning like in 1b) this follows from $\langle s'_{\xi(s,m,i)}, m'_{\xi(s,m,i)}, es\rangle \asymp os$ and the fact that $m'_{\xi(s,m,i)} = m_o$ is blocked.

□

5 Conclusions.

We have written a mathematical specification of a variant of a standard profile for smart cards. This specification was destined from the beginning to be completely formalized in Type Theory (using the proof assistant Coq). We have

presented it in a way such as to make it understandable to anyone used to ordinary mathematical language (i.e. based upon ordinary set theory).
The specification possesses a high level of abstraction. For instance, we have introduced the directory structures as an abstract data type with a restricted set of necessary properties and specified the instructions of the card by means of pre- and post-conditions instead of introducing particular algorithms (functions) operating on concrete data structures.

The code produced and verified with the assistance of Coq embodies the complete formalization of the operating system's mathematical specification as well as the proofs of the properties described in the preceding section. The completely formalized versions of both the specification and the proofs were almost directly generated from the mathematical specification. The components that were left abstract in the mathematical specification appear as parameters of the Coq code. For the proof of the main proposition, which requires to reason over coinductively defined data, we made use of the tactic `Cofix`. We refer to [7] for a detailed presentation of this tactic.
The whole code amounts to nine Coq files. It took about 20 man-hours to write down and check the files that constitute the specification whereas the proofs were carried out in no more than 4 hours. We also generated an HTML documentation of the code, using a tool provided by the assistant. This documentation is available at `www.fing.edu.uy/~mf/SmartCards`.
The concrete profile to be specified was chosen as an initial test case for the applicability of type theory to the specification of smart cards' operating systems. Given the resulting process and outcome, we believe that the approach can be scaled up to more complex variants within reasonable margins of effort.

A formal specification like the one obtained can have several uses:

- It can be tested by mechanically deriving and verifying proofs of required or expectable properties. In this work, we have given one example of this. Another example of such a property would be that no file operations can be executed until a successful PIN verification has occurred. To prove this of the given specification requires to provide a particular automaton that implements the mentioned requirement. Nevertheless, a simple modification could be introduced so as to impose the property in question without making use of an automaton, namely to require as a pre-condition for every file operation that the error counter be equal to 0. Other properties that can be proved of the present specification are those concerned with the result of sequences of file operations, e. g. about the result of reading operations carried out after updating operations. These follow quite straightforwardly from the specification of the instructions.
- It can serve as the basis from where to obtain formally certified implementations written in machine code. One way to achieve this is to verify machine code specifications against the original one. We think that the tools described in [6] can be useful in this respect. Another way would be to derive a func-

tional prototype satisfying the specification and to use the prototype in turn to obtain the machine code implementation, possibly via some machinery of program transformations. In this sense it would be interesting to point out that, according to our estimations, the task of deriving a correct prototype can be carried out in a relative small amount of time. This would require to define implementations for the components that have been abstractly specified, as well as to implement a particular automaton (i.e. application) for the card.

- It can serve for specifying standards of smart cards' operating system. This is particulary relevant at the present state of the art of this field, where no standard has imposed itself yet. In this respect, a formal specification would first of all be useful to detect errors in the informal specifications. On the other hand, the formal specification of standards is essential for the introduction of formal methods in the processes of construction of the cards. And, finally, even if traditional methods of constructions are to continue in use, a formal specification would serve as 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.

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, following any of the approaches referred to above. We have also started investigating a particular class of smart cards, those known as *Java Cards* [12]. These cards admit their applications being completely written in the Java language [8], 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

1. B. Barras et al. The Coq Proof Assistant Reference Manual, Version 6.1. Rapport de recherche 203, INRIA, 1997.
2. D. Bolignano. Towards a Mechanization of Cryptographic Protocol Verification. In 9th. International Computer-Aided Verification Conference, June 1997.
3. D. Bolignano. Towards the Formal Verification of Electronic Comerce Protocols. In 10th. Computer Security Foundations Workshop, June 1997.
4. 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.
5. T. Coquand. *Une Théorie des Constructions.* Thèse de doctorat, Université Paris 7, 1985.
6. J.C. Filliatre. Proof of Imperative Programs. In *Chapter 18 of* The Coq Proof Assistant Reference Manual, Version 6.2.4, 1999.
7. E. Giménez. A tutorial on recursive types in coq. Technical Report 0221, INRIA, 1998.
8. J. Gosling, B. Joy, and G. J. Steele. *The Java^{TM} Language Specification.* Addison-Wesley, 1996.

9. P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.
10. 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.
11. W. Rankl and W. Effing. *Smart Card Handbook.* John Wiley & Sons, 1997.
12. Sun Microsystems, Inc. Java Card Technology, 1999. Available in `http://java.sun.com/products/javacard/`.