Formalisation and Verification of the GlobalPlatform Card Specification Using the B Method

Santiago Zanella Béguelin

INRIA Sophia Antipolis,
2004 Route des Lucioles, 06902 Sophia Antipolis, France
Santiago.Zanella@inria.fr

Abstract. We give an overview of an application of the B method to the formalisation and verification of the GlobalPlatform Card Specification. Although there exists a semi-formal specification and some effort has been put into providing formalisations of particular features of smart card platforms, this is, as far as we know, the very first attempt to provide a complete formalisation. We describe the process followed to synthesise a mathematical model of the platform in the B language, starting from requirements stated in natural language. The model consistency has been thoroughly verified using formal techniques supported by the B method. We also discuss how the smart card industry might benefit from exploiting this formal specification and outline directions for future work.

1 Introduction

1.1 Smart Cards

Smart cards [1] are small portable devices, usually the size of a credit card, embedded with either only a memory chip or with both a microprocessor and a memory chip. They are capable of communicating with an external network terminal through a card reader and a contact or contact-less interface by exchanging Application Protocol Data Unit (APDU) messages. Smart cards are broadly used in a significant number of applications, ranging from telecommunications, transport and access control to electronic purses and e-government. Most popular applications include debit cards, prepaid phone cards, and the Subscriber Identity Module (SIM) cards used in mobile phones to hold subscriber's personal information and settings.

Early smart card applications were written for a specific combination of operating system and hardware and designed to run as the sole application in a card. This scheme forced card issuers to commit to a particular implementation without any possibility for post-issuance modification and at the same time compelled users to carry a different card for each application they wished to use. The need to overcome these difficulties lead to the concept of multi-application

smart cards, capable of hosting multiple applications and allowing applications to be loaded, upgraded and unloaded after issuance. Multi-application smart cards became a reality thanks to the increase in the computational power and memory capacity of the cards and the development of general purpose card operating systems in the last decade, including *Java Card*, *MultOS*, *Windows for Smart Cards* and *BasicCard*. Each of these operating systems provides a common development framework and standard programming interfaces that improve the portability of developed applications across different card implementations and enable multiple applications to coexist on a single card sharing services and data. This increased flexibility, unfortunately, does not come for free, since it brings up new security concerns that need to be addressed.

In spite of all its benefits, the adoption of multi-application smart cards had been slowed down due to the absence of standards for the security and application management aspects of smart card platforms until the GlobalPlatform consortium published their specifications, currently accepted as de facto industry standards. However, because GlobalPlatform specifications are expressed in natural language using a semi-formal notation, they are subject to misinterpretations and their consistency cannot be formally verified. The following excerpt from [2, Sect. 6.9.1.1] describing the state transitions of the Cardholder Verification Method (CVM), might help in understanding the level of detail of the natural language specification:

> "At the end of a Card Session the CVM state shall transition back to Active, except if the CVM state transitioned to the CVM state Blocked during the Card Session".

What should happen if the CVM state transitions to Blocked, but is later unblocked and then transitions to another state like Validated (meaning successful cardholder authentication) during the same card session? Common sense dictates that the CVM state should nevertheless be reset to Active. A careless reader may understand exactly the opposite.

1.2 GlobalPlatform

GlobalPlatform (GP) is a nonprofit organisation established in 1999 by leading companies from the industry, the government sector and vendor community whose goal is to establish and drive the adoption of standards to enable an open and interoperable infrastructure for smart cards, devices and systems that simplifies and accelerates the development, deployment and management of applications across industries.

The main assets of GlobalPlatform are their specifications, available royalty-free and downloadable from their website [3]. GlobalPlatform specifications cover the card itself as well as their associated devices and systems and are applicable to both single and multi-application scenarios. By providing these specifications

on a royalty-free basis GlobalPlatform succeeded in promoting their acceptance as standards and in accelerating the adoption of smart card technology. An increasing number of card vendors and application developers are adopting GlobalPlatform specifications as the standard upon which to base their smart card infrastructures. Present estimates indicate that the number of GlobalPlatform compliant smart cards in circulation exceeds 670 million (600 million of which are SIM cards).

1.3 Paper Overview

This work gives an overview of a formal specification of the GlobalPlatform security and application management architecture using the B method. This formal model provides an abstract reference specification expressed using a formal mathematical language that has the potential for eliminating any ambiguity that may remain in the existing semi-formal documentation expressed in natural language. The model also provides a general framework from which other participants may build up and share their contributions.

The remainder of the paper is structured as follows: Section 2 gives a general overview of the B method, Sect. 3 introduces the semi-formal specification provided by GlobalPlatform while Sect. 4 describes its formalisation. Section 5 shows using an example how the formalisation should be interpreted, Sect. 6 describes the formal proof process and finally, Sect. 7 concludes presenting related research on the subject and future work.

2 The B Method

2.1 Overview

The B method is a model-oriented formal method for engineering software systems developed by Abrial [4]. It is not only a notation for specifying systems, it is a comprehensive formal method that covers the entire software development cycle: from requirements specification to code generation. The method is based on the mathematical principles of set theory and predicate calculus while its semantics is given using a variant of Dijkstra's weakest precondition calculus [5].

A B specification is composed of a hierarchy of components that are described using the Abstract Machine Notation (AMN). AMN greatly resembles the notation used in high-level imperative programming languages and provides the representation and manipulation of mathematical objects such as natural numbers, sets and functions. The notation supports typical logical and set-theoretical operators as well as some other useful operators that simplify the manipulation of complex mathematical objects such as functions and relations (see Table 1 for a description of the most common operators).

Table 1. Commonly used B operators

Notation	Semantics
$\mathcal{P}(X)$	Set of all subsets of X
$X \times Y$	Cartesian product of the sets X and Y
$X \leftrightarrow Y$	Set of relations of X to Y, or equivalently $\mathcal{P}(X \times Y)$
$X \nrightarrow Y$	Set of partial functions from X to Y
$X \rightarrow Y$	Set of total functions from X to Y
$X \rightarrowtail Y$	Set of partial injective functions from X to Y
$\mathsf{Id}(X)$	Identity relation on X
R^{-1}	Inverse relation of R
$\mathsf{dom}(R)$	Domain of the relation R
$\mathsf{ran}(R)$	Range of the relation R
$R[X]$	Relational image of X under the relation R
$X \lhd R$	Binary relation R restricted to pairs with first component in X
$X \lhd\!\!\!\!- R$	Binary relation R restricted to pairs with first component not in X
$R \rhd X$	Binary relation R restricted to pairs with second component in X
$R \leftarrow\!\!\!\rhd S$	Relation R overridden by S. Equivalent to $(\mathsf{dom}(S) \lhd\!\!\!\!- R) \cup S$
$R \otimes S$	Direct product. Defined as $\{x, (y, z) \mid x, y \in R \wedge x, z \in S\}$

Each component in a specification represents nothing but a state machine: a set of variables defines its state and a set of operations – state transitions – forms an interface used to query and modify that state. Variable types and additional constraints on the variables are introduced as invariants of a machine. State transitions in AMN are specified by means of *generalised substitutions*. A generalised substitution is a construct built up from basic substitutions, such as $x := e$, corresponding to simple assignments to state variables. The simultaneous substitution $(S_1||S_2)$, the bounded non-deterministic choice (**CHOICE** S_1 **OR** S_2 **END**), and the sequential composition $(S_1; S_2)$ are examples of constructors used to build up generalised substitutions from simpler ones. There exist three different types of components:

Abstract Machines Top-level components in specifications that describe state machines in an abstract way, perhaps using non-deterministic state transitions. They do not need to be directly implementable. Figure 1 shows the typical structure of an Abstract Machine.

Refinements Enriched versions of either an Abstract Machine or another Refinement. They must preserve the interface and behaviour but may otherwise reformulate the data and operations of the original machine. The variables in the original machine may be either preserved or refined in terms of new variables. The relationship between the original and the refined variables is

stated as an invariant of the Refinement.

Implementations Ultimate step in the refinement of an Abstract Machine, both data and operations need to be implementable in a high-level programming language. As a consequence, non-deterministic substitutions or abstract variables (e.g. relations, functions) are not allowed in Implementations. An Implementation may rely on the operations and data imported from Abstract Machines.

MACHINE M
SEES
Constituents of Abstract Machines referred to here can be accessed in a read-only fashion
SETS
Given sets. A given set is introduced by its name and an optional enumeration of its values and may be used to type variables and constants
CONSTANTS
Constants that can be referred to in a read-only way
PROPERTIES
Properties of given sets and constants. Constants must be typed here
VARIABLES
State variables
INVARIANT
Variable typing and additional constraints on the machine variables
INITIALISATION
Assignment of initial values to the machine variables
OPERATIONS
Definition of machine operations
END

Fig. 1. General structure of an Abstract Machine. A short description follows each clause

Operations are made up of a header and a body. The header of an operation is an identifier, designating its name, optionally followed by a parenthesised comma-separated list of input formal parameters. A list of output parameters may be specified preceding the name of the operation. The body of an operation is a generalised substitution. An operation that has input parameters is written using a precondition substitution (**PRE** P **THEN** S **END**) that types its input parameters and may express other properties that shall hold at the time the operation is executed.

A B model can be mechanically syntax and type checked. Thanks to the mathematical semantics of the method, a B model may also be subject to formal proof to verify its consistency (including the preservation of invariants) and the correctness of all refinement steps.

2.2 Tool Support and Industrial Applications

There are currently two commercially available toolkits that support the complete development of systems using the method, *Atelier B* from ClearSy, and *B-Toolkit* from B-Core. During the development of this specification we opted to use the Atelier B toolkit which allows to automatically type check and verify the syntax of a specification as well as generate the proof obligations that once discharged guarantee its consistency. To discharge these proof obligations Atelier B provides a theorem prover which can be run either in automatic or interactive mode. A survey of available tools based on the B method can be found in [6].

The B method is particularly suited to support the development of safety-critical systems. It has been successfully applied in large industrial projects as the Meteor automated subway in Paris [7] and the IBM Customer Information Control System (CICS) [8]. It is also commonly accepted in the smart card field as a suitable method for formalising and verifying applications [9, 10] or particular aspects of smart card platforms [11–13]. For further details on the method, the reader is encouraged to refer to textbooks such as [4] or [14].

3 GlobalPlatform Semi-formal Specification

The functional and security related requirements for GlobalPlatform cards are specified in a semi-formal way in [2] and [15]. In some aspects both specifications overlap, and this is source of inconsistencies. The functional requirements of GlobalPlatform compliant cards, including all card content management functions (e.g. application installation and deletion) and their runtime behaviour, are described in [2] by means of natural language and conceptual diagrams, while [15] describes in detail, using a semi-formal notation, the security requirements of the platform, including requirements for the underlying Runtime Environment (RTE), Operating System and Integrated Circuit. These requirements are expressed in terms of a number of *Security Features* (SF) which are themselves specified in terms of one or more tables.

The header of a SF table (its first row) states the precondition that triggers its activation. Each following row except the last one describes the rules by which a user is permitted to perform some operation on some object in the card. For example, the unique row in the body of Table 2 describes the conditions that an incoming command in the APDU buffer must meet in order to be accepted for processing. The last row of the table states its postcondition – the actions to be taken in response to the operations.

SF tables are linked together by their preconditions and postconditions as shown in Fig. 2. A glance at the figure should suffice to justify the need to specify in a precise manner the interrelationships between tables. The number of tables and the greatly tangled dependencies among them makes a natural language

Table 2. A SF table adapted from [15] describing the validation of incoming APDU commands

Precondition:	The Platform Code has control. AN APDU message is received.		
Short Form:	OP_alive	**Link(s) back:**	Table 5–34: The Supervisor Security Feature (Invocation of security mechanisms)
Operation	**Object(s)**	**Security Attribute(s)**	**Rule(s)**
Any APDU command	APDU Buffer [Command]	Command[CLA] Command[INS] Command[Parameters] GP Registry[Selected App]	(if the GP Registry[Selected App] is the ISD[AID] or any other SD[AID]) or (if the Command is Select[Any App]) then ((the Command[CLA] and Command[INS] shall be included in the card configuration) and (the Command[Parameters] shall not be illegal, missing, unexpected, out of range or have out of range lengths))
Result (rule evaluates to true):	The command is accepted for processing. This Table links to Table 5–37: The Supervisor Security Feature (Command dispatch) to dispatch the command.		
Result (rule evaluates to false):	An appropriate GPCS error APDU response message is returned to the off–card entity.		

specification error-prone.

A particular SF, the Supervisor SF acts as the starting point for all card operations. The execution of a card operation may be interpreted as follows:

1. The Supervisor SF table is invoked, its postcondition links to another table depending on the operation type;
2. In the new table, the rule field in the row corresponding to the operation is evaluated. According to the result, the table postcondition may link to another table or may terminate the execution of the operation;
3. The previous step is repeated until the execution terminates. Any changes in the card state are committed upon completion of the execution.

4 Formalisation of the GlobalPlatform Specification

A standard specification for such critical functions as the security and card management architecture of a card platform must be carefully designed, validated and verified in order to obtain a maximal level of confidence in its implementations. Stating and structuring the specification in natural language, by means of tables or diagrams is a good starting point. However, natural language specifications are error-prone, subject to misinterpretations and cannot be formally verified. The obvious step to achieve a higher level of reliability is to derive a formal specification and apply formal techniques to verify its consistency and desirable properties. The application of formal methods is also a must for developers seeking the highest Common Criteria evaluations assurance levels (i.e. those from EAL5 to EAL7).

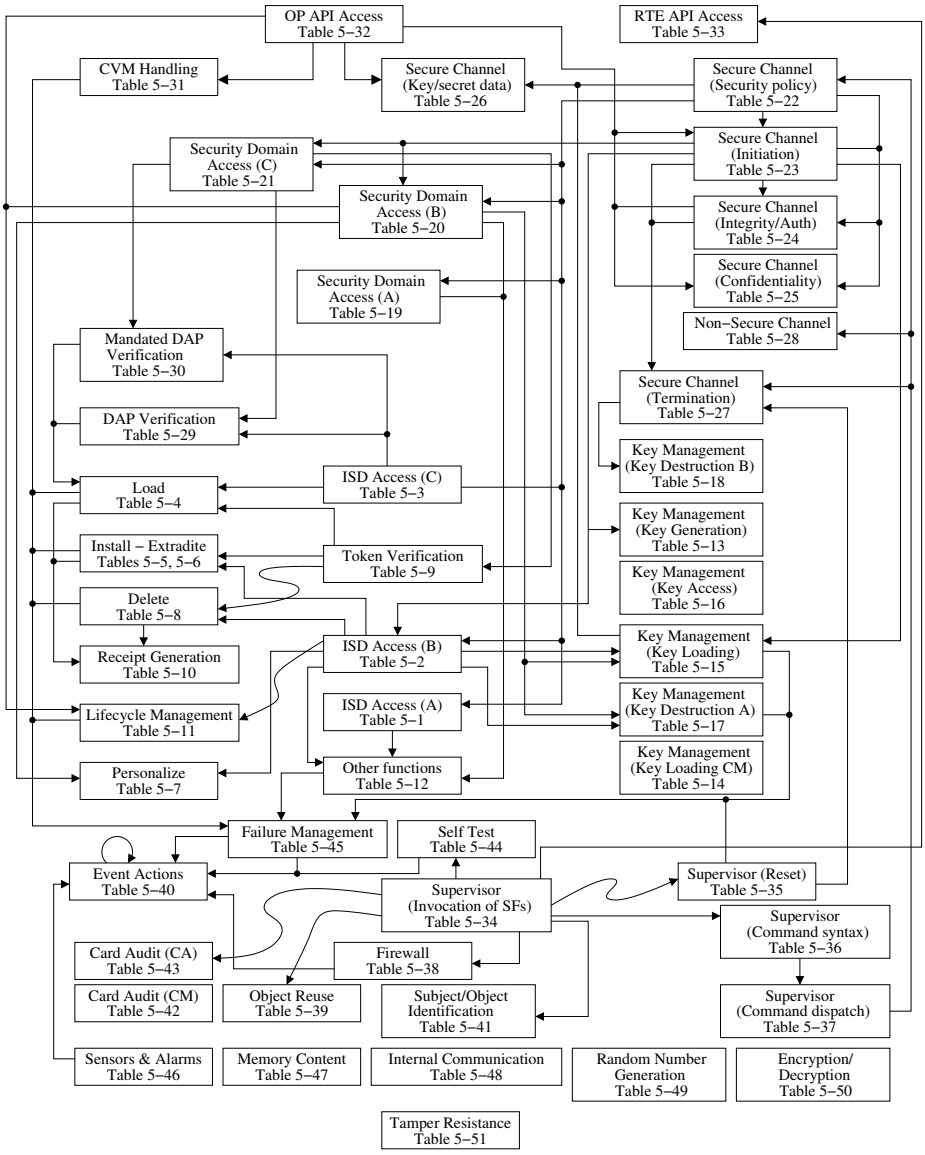

Fig. 2. Pre- and postcondition links between SF tables as appearing in [15]. The postcondition of each table at the tail of an arrow establishes the precondition of the table at the head of the arrow. Tables without incoming arrows are called *Function Tables* and are *activated* on demand

In the rest of this section we describe the formalisation of the GlobalPlatform specification. A royalty-free complete and commented version of the formal model may be obtained from the GlobalPlatform website [3].

4.1 Specification Architecture

Figure 3 shows a view of the specification architecture where arrows represent composition links and boxes represent components. The specification is organised in four layers of increasing detail according to their model of the card state. Each layer except the lowest one is represented by an Abstract Machine and its Implementation and each Implementation relies in turn on the Abstract Machine in the next lower layer in the hierarchy. This hierarchical model facilitates the construction of the specification and its formal verification. A short description

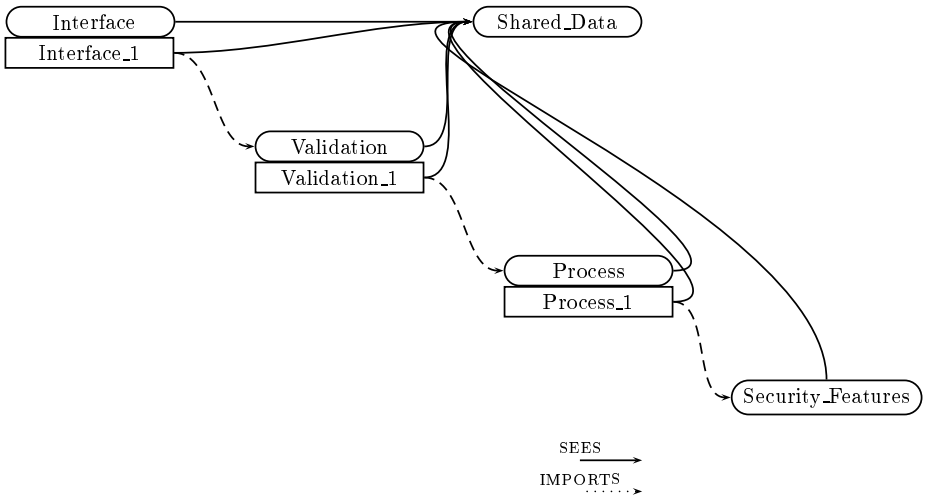

Fig. 3. Overall specification architecture. An arrow pointing from component *M* to component *N* should be read as *M* **SEES**|**IMPORTS** *N*. Ovals represent Abstract Machines, boxes Implementations

of each Abstract Machine is as follows:

Shared_Data Definitions of sets and constants shared among different components of the specification. Some of these sets and constants are used as types (e.g. *AID*, the set of valid application identifiers), some are used as configuration parameters for the card (e.g. *ISD*, the Issuer Security Domain AID) and others as abstract functions (e.g. *select_aid*, that extracts the AID of the application to be selected from a SELECT APDU command).

Interface Outlines the execution of APDU commands and the behaviour upon Card Reset with respect to the set of currently open logical channels.

Validation Models the validation, dispatching and processing of APDU commands as well as the behaviour upon Card Reset or Power up. Each operation deals with a different type of interaction, a Card Reset, a Power Up event or a specific type of APDU command. The card state is extended to include the currently selected application and all applications in the GP Registry.

Process Security Features that deal with the validation, dispatching and security processing of APDU commands as well as GP API methods are first introduced in this machine. The card state is extended to represent most of the card features including registered load files, the CVM and cryptographic keys. The actual processing of APDU commands is specified in detail.

Security_Features Security Features as defined in [15] are represented as operations in this component. Operations in the *Process* machine make use of this component to ensure conformance to the selected security policies.

The Security Features are described in [15, Chapter 5] in terms of attempted operations performed on objects. The outcome, i.e. whether the attempt succeeds or fails, is decided by a set of rules which are expressed in terms of security attributes. Objects and security attributes identified in [15] are represented by the variables and constants of the machine specifications. The execution of an APDU command is sketched in the *Interface* machine. The state of the card in this machine is only represented by the set of currently open Logical Channels and so there is no need to distinguish among the different types of APDU commands at this stage. In the *Validation* machine the card state is extended to include information about the registered applications and security domains, their life cycle and the currently selected application. The *Interface_1* Implementation describes how the operations in the *Interface* machine are implemented using the operations in the *Validation* Abstract Machine.

The level of detail in each stage depends on how abstract is the representation of the card state: how many and which variables are used to represent it. As the card state is extended in the lower layers, the specification becomes richer and more detailed. Ultimately, all operations are implemented on operations in the *Security_Features* Abstract Machine, meaning – once the specification is proved correct – that the functional requirements of the platform are implementable in terms of the Security Features and satisfy the security policies enforced on the platform.

5 An Example: Installing a New Application

Smart cards employ APDU messages for carrying out the communication with card terminals. An APDU contains either a command message (Table 3) sent

from the terminal to the card, or a response message sent from the card to the terminal. The communication is half-duplex and follows a master-slave model. The smart card waits for APDU commands from the terminal in its interface. Once a command is received, the card executes it and sends back a response APDU message.

Table 3. APDU command message structure

Field	**Description**	**Length**
CLA	Class Byte	1 byte
INS	Instruction Byte	1 byte
P1	Reference Control Parameter P1	1 byte
P2	Reference Control Parameter P2	1 byte
Lc	Data Length	1 byte
Data	Command Data	Variable
Le	Length of Expected Data	1 byte

Navigation through the execution of an APDU command can be accomplished by following in the specification the implementation path from the *Interface* machine to the *Security_Features* machine. If the process is stopped somewhere in between, the result would be an abstract specification of the command behaviour. We illustrate how to obtain a specification of the execution of an INSTALL[FOR INSTALL AND MAKE SELECTABLE] command issued by the Card Administrator in a simplified scenario. The command requests the installation of an application from an executable module and sets its life cycle state to SELECTABLE, enabling the application to be selected and receive commands from off-card entities. The executable module from where the application is instantiated must be present within and executable load file in the card. The structure of a correctly formatted INSTALL[FOR INSTALL AND MAKE SELECTABLE] command is shown in Table 4.

The reception of an APDU command is represented by the execution of the **APDU** operation in *Interface* (Fig. 4), the input parameters being the command fields. Supposing the channel information contained in the CLA byte of the command corresponds to a currently open logical channel, the specification mandates some status word to be returned and allows the set of open logical channels to be modified. Considering that the card state in this machine is restricted to the set of open logical channels, this is a complete specification of the APDU outcome with respect to this representation.

The implementation of the **APDU** operation in *Interface_1* (Fig. 5) discriminates between different commands, and delegates the processing of the command

Table 4. INSTALL[FOR INSTALL AND MAKE SELECTABLE] command content

Field	Content
CLA	CLA_PROPRIETARY/CLA_SPROPRIETARY
INS	INS_INSTALL
P1	P1_INSTALL_SELECTABLE
P2	NULL
Lc	Data Length
Data	Executable Load File AID, Executable Module AID, Application AID, and Application Privileges
Le	NULL

$sw \leftarrow$ **APDU**(CLA,INS,$P1$,$P2$,$Data$,Le) =
PRE
$CLA \in$ BYTE $\wedge$ $INS \in$ BYTE $\wedge$ $P1 \in$ BYTE $\wedge$
$P2 \in$ BYTE $\wedge$ $Data \in$ DATA $\wedge$ $Le \in$ BYTE
THEN
IF $channel(CLA) \notin open\_channels$ **THEN**
$sw :=$ SW_ERROR
ELSE
$sw :\in$ STATUS_WORD ||
$open\_channels :(open\_channels \subseteq$ LOGICAL_CHANNEL $\wedge\ 0 \in open\_channels)$
END
END

Fig. 4. APDU operation in the *Interface* machine. The notation $v : (P)$ should be read as 'v *becomes such that* P *holds*'. $v :\in S$ assigns to v any value in the set S

to the **Install_For_Install** operation in the *Validation* machine (Fig. 6).

Assuming that the command syntax is correct, the card life cycle is not TERMINATED ($app\_life\_cycle(ISD) \neq TERMINATED$), the command data is valid and the off-card entity is authenticated (AUTHENTICATED $\in sl$), the **Install_For_Install** operation restricts the modification of the card state, but does not determine exactly how the state is modified. Up to this point, we have obtained an abstract description of the functional and security requirements for the command.

The implementation of the **Install_For_Install** operation makes use of the **Install_For_Install_1** operation in *Process* (Fig. 7) to describe the actual processing of the command. If the card is not locked, the executable module from where the application is to be instantiated exists, and the AID and privileges assigned to the application would not leave the card in an inconsistent state, an entry is created in the registry for the application, its associated security do-

```
sw ← APDU (CLA,INS,P1,P2,Data,Le) =
  VAR ch,bb IN
    ch := channel(CLA);
    bb ← IsOpen(ch);
    IF bb = FALSE THEN
      sw := SW_ERROR
    ELSE
      sd ← IsSDSelected(ch);
      IF sd = TRUE THEN
        CASE CLA OF
          EITHER CLA_PROPRIETARY,CLA_SPROPRIETARY THEN
            CASE INS OF
              EITHER INS_INSTALL THEN
                CASE P1 OF
                EITHER P1_INSTALL_SELECTABLE THEN
                  sw ← Install_For_Install(ch,CLA,INS,P1,P2,Data,Le)
                ...
```

Fig. 5. Implementation of the **APDU** operation in *Interface_1*

```
sw ← Install_For_Install(ch,CLA,INS,P1,P2,Data,Le) =
  PRE
    ch ∈ open_channels ∧
    CLA ∈ {CLA_PROPRIETARY, CLA_SPROPRIETARY} ∧
    INS = INS_INSTALL ∧ P1 = P1_INSTALL_SELECTABLE ∧ P2 ∈ BYTE ∧
    Data ∈ DATA ∧ Le ∈ BYTE ∧ selected(ch) ∈ security_domains
  THEN
    IF
      P2 = NULL ∧ Le = NULL ∧ app_life_cycle(ISD) ≠ TERMINATED ∧
      data ∈ VALID_INSTALL_FOR_INSTALL_DATA ∧ AUTHENTICATED ∈ sl
    THEN
      sw :∈ STATUS_WORD ||
      applications, security_domains, app_life_cycle, default_selected :(
        applications ⊆ AID ∧ applications$0 ⊆ applications ∧
        security_domains ⊆ applications ∧ security_domains$0 ⊆ security_domains ∧
        app_life_cycle : applications → LIFE_CYCLE ∧
        app_life_cycle$0 ⊆ app_life_cycle ∧
        default_selected ∈ applications)
    ELSE
      sw := SW_ERROR
    END
END
```

Fig. 6. Install_For_Install operation in *Validation*. Observe how modifications to the card state are restricted using the *becomes such that* substitution. The value of a variable prior to the substitution is referenced by appending $0 to its name

main, life cycle state and privileges. This last **Install_For_Install_1** operation is implemented using only the operations in *Security_Features*, which correspond to the Security Features described in [2].

```
sw ← Install_For_Install_1(ch,CLA,INS, P1,P2,Data,Le) =
 PRE
  ch ∈ open_channels ∧
  AUTHENTICATED ∈ security_level(ch) ∧
  app_life_cycle(ISD) ≠ TERMINATED ∧
  ...
 THEN
  IF
   app_life_cycle(ISD) ≠ CARD_LOCKED ∧
   mod_aid(Data) ∈ executable_modules ∧
   app_aid(Data) ∉ applications ∧ app_aid(Data) ∉ executable_load_files ∧
   (pr_default_selected ∈ privileges(Data) ⇒
    default_selected = ISD ∧ app_life_cycle(ISD) ≠ OP_READY)
  THEN
   sw := SW_OK ||
   applications := applications ∪ {app_aid(Data)} ||
   app_sd(app_aid(Data)) := elf_sd(mod_elf(mod_aid(Data))) ||
   app_elf(app_aid(Data)) := mod_elf(mod_aid(Data)) ||
   app_privileges := app_privileges ∪ {(app_aid(Data), privileges(Data))} ||
   app_life_cycle(app_aid(Data)) := SELECTABLE ||
   IF pr_default_selected ∈ privileges(Data) THEN
    default_selected := app_aid(Data)
   END
 END
```

Fig. 7. Install_For_Install_1 operation in *Process*

The example above may not give much insight into the dimension of the specification. Every APDU command and GP API function is specified like the the INSTALL[FOR INSTALL AND MAKE SELECTABLE] command just described. The complete specification being around eight thousand lines long is not a trivial case study.

6 Formal Proof

The B method semantics allows to mechanically generate the *Proof Obligations* (PO) to be discharged in order to guarantee that the model is mathematically consistent. The PO may come from the need to prove different kinds of properties:

Initialisation consistency Assuming the stated properties of constant and sets, the initial state of an abstract machine must be established by the gen-

eralised substitution under its **INITIALISATION** clause;

Invariant preservation As state transitions in the B language are specified via operations, and transitions shall not violate the invariant, each operation in an abstract machine must preserve the invariant;

Refinement correctness The **INITIALISATION** substitution and each operation in a refinement shall fulfil the specification of their abstract versions.

With the help of the Atelier B automatic prover almost 90% of the generated PO were proved automatically, the remaining obligations were proved interactively. This gives a complete guarantee of the model consistency assuming the correctness of the tool and the underlying theory. As we have nothing but a natural-language specification to compare it with, the specification correctness cannot be verified. However, simple invariants can be proved to gain more confidence. Some interesting invariants may be specified in the *Process* machine. For example, that the ISD should be the default selected application when the card is in the OP_READY state

$$card\_life\_cycle = \text{OP\_READY} \Rightarrow default\_selected = ISD \ ,$$

or that selected applications shall not be in the INSTALLED state

$$selected \rhd app\_life\_cycle\_state^{-1}[\{\text{INSTALLED}\}] = \{\} \ .$$

In fact, trying to prove the following invariant

$$elf\_sd \in executable\_load\_files \rightarrow security\_domains$$

that ensures that every load file is associated with a security domain, uncovered an omission in the original specification of the DELETE command runtime behaviour that allows a security domain to be deleted even if it has executable load files associated.

Table 5 gives a summary of the formal proof of the specification. Due to the way the specification is constructed most of the PO arise in proving that an implementation is correct with respect to its abstract counterpart than in proving that an Abstract Machine operation does not violate an invariant. The former involves proving that every possible behaviour allowed by the implementation is allowed by the original operation and that the result in terms of the implementation variables is the same, as well as proving that the preconditions of the operations on which the implementation relies are satisfied. The later amounts to proving that the operation does not violate the machine invariant. In contrast, in this specification proof obligations of invariant preservation tend to be more complicated than proof obligations of implementation correctness. As a result, the layered architecture of the specification helps to reduce the number of non-trivial proof obligations but increases the total number of proof obligations.

Table 5. Proof summary

Component	Proved Interactively	Proved Automatically	Total
Interface	0	2	2
Interface_1	11	211	222
Validation	3	161	164
Validation_1	144	316	460
Process	54	841	895
Process_1	118	987	1,105
Security_Features	55	781	836
Total	385	3,299	3,684

Much of the proof obligations generated in a component tend to be very similar. In many cases, the same proof script may be used to discharge several proof obligations. In this way, the number of actual interactive proofs is greatly reduced. There were no hard proofs, and no need to develop theories for the proof assistant with the exception of a few set theory lemmas.

7 Conclusion

While multi-application platforms have long been seen as the future of smart cards, the lack of commonly trusted standards for application management and their security concerns has slowed their deployment. We strongly believe that this work will help improve the confidence in GlobalPlatform specifications and accelerate their acceptance as trusted standards. The objective of providing a formal model of the GlobalPlatform specifications was successfully achieved in 7 men months, a neglectable cost considering the payback. As the work was in progress, omissions and inconsistencies were detected in the original specifications, including one that could lead to the execution of unauthorised APDU commands when the card is in the TERMINATED life cycle. Some of these issues were resolved in fluent contact with GlobalPlatform, while others still remain to be settled by the GlobalPlatform Card Specification Workgroup. The resulting exchange of opinions is an invaluable documentation that gives the rationale behind the decisions taken to resolve those issues. We expect most of these documentation to be included in the next release or amendment of the GlobalPlatform specifications.

7.1 Future Work

Test Automation. Reduced time-to-market is critical in the smart card industry and testing is a bottleneck for the deployment of new card platforms. The availability of the formal model opens the way to specification-driven test automation. This means that test cases can be generated, executed and assessed

automatically using the formal specification, speeding time-to-market for new developments. Furthermore, the coverage of the generated tests may be measured against the specification using rigorous techniques. There exist actual tools that support test automation based on B specifications [16, 17] and there is at least one tool developer investigating the possibility of using our formal model to automate tests for GlobalPlatform compliance.

Formal Development. The B method supports fully formal software development. An interesting line of work is to investigate if the model could be, at least partially, refined down to executable code. Another possibility is to refine the existing model to specialise it for particular card configurations: proving the refinement correctness amounts to justifying compatibility with the specifications. The layered structure of the specification makes the model easy to extend.

Reference Implementation. Instead of refining the formal model to executable code, an alternative approach is to reuse the model to derive a reference implementation annotated in a specification language like JML, together with a justification that the implementation satisfies the specification. Such annotated implementation may be subject to model checking and static verification.

Specification Maintenance. The smart card field is highly dynamic, specifications must evolve to satisfy the market requirements and this formal model is not an exception. Future versions of GlobalPlatform specifications are already scheduled for 2006. Rather than becoming a load, both specifications may benefit from evolving simultaneously, envisaging the possibility of a future convergence.

7.2 Related Work

Formal methods had been applied to the verification of real-world smart card applications. Significant research effort has been put into the formalisation of specific smart card platform implementations. However, most of the work has been concerned with the Java Card platform (e.g. virtual machine, bytecode verifier and API). We detail some of the main achievements below.

Application Verification Stepney et al. [18] give a specification and formal proofs of some security properties of an industrial strength electronic purse application using Z. Huisman and Cataño [19] use ESC/Java to annotate with functional specifications and statically verify the code of an electronic purse Java Card applet. The KeY tool [20] is an interactive theorem prover based on Dynamic Logic for Java Card source code annotated in OCL. Krakatoa [21] and Jack [22] are tools for the verification of JML-annotated Java Card programs, using the Coq proof assistant. Jack may also generate proof obligations for other theorem provers like PVS and Simplify.

Java Card Virtual Machine (JCVM) The VerifiCard [23] project succeeded in giving complete formalisations of the Java Card platform implementation at both bytecode and source code level. A partial formalisation of the JCVM using the B formal method is given in [9]. The Bali project [24] formalises in Isabelle/HOL a large body of the Java platform, including operational semantics for the source and bytecode languages and an abstract ByteCode Verifier (BCV); [25] provides executable Coq specifications for the JCVM as well as a BCV. [26] is a volume dedicated to the formal syntax and semantics of Java. All these works provide means to reason formally about applications written in the Java Card programming language and enable the verification of applet correctness.

Java Card API Interface specifications for the Java Card API have been written in the JML and ESC/Java specification languages and are presented in [27–29]. [30] is a recent overview of JML tools and applications. The LOOP (Logic of Object-Oriented Programming) tool was used to verify that the actual Java Card API classes deployed in smart cards satisfy the JML interface specifications [31].

Protocols Sabatier and Lartigue present their result on the validation of the transaction mechanism for smart cards using the B method in [11]. A semi-formal and a formal B specification of the T=1 protocol used to transfer messages between a smart card and a reader is presented in [12]. This approach complements our work, since we only deal with on-card features.

Acknowledgements

I would like to thank Gilles Barthe for his helpful comments on preliminary versions of this paper. Jean-Louis Lanet and Lilian Burdy kindly provided their expertise in the B method and valuable insights while the specification was being developed. Marc Kekicheff from GlobalPlatform promptly provided clarifications on the semi-formal specifications when needed.

References

1. Rankl, W., Effing, W.: Smart Card Handbook, second edition. John Wiley & Sons, Inc. (2000)
2. GlobalPlatform: Card Specification. Version 2.1.1. (2003)
3. GlobalPlatform. `http://www.globalplatform.org`.
4. Abrial, J.R.: The B Book - Assigning Programs to Meanings. Cambridge University Press (1996)
5. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Upper Saddle River, NJ, USA (1976)
6. Site B Grenoble. `http://www-lsr.imag.fr/B/b-tools.html`.
7. Behm, P., Benoit, P., Faivre, A., Meynadier, J.M.: METEOR: A successful application of B in a large project. In: Proceedings of FM'99: World Congress on Formal Methods. (1999) 369–387

8. Hoare, J., Dick, J., Neilson, D., Sorensen, I.: Applying the B technologies to CICS. In: Proceedings of FME '96: Industrial Benefit and Advances in Formal Methods. Third International Symposium of Formal Methods Europe. (1996) 74–84
9. Lanet, J.L., Requet, A.: Formal proof of smart card applets correctness. In Quisquater, J.J., Schneier, B., eds.: Third Smart Card Research and Advanced Application Conference, Louvain-la-Neuve, Belgium (1998)
10. Bert, D., Boulm, S., Potet, M.L., Requet, A., Voisin, L.: Adaptable translator of B specifications to embedded C programs. In Araki, I.K., Gnesi, S., Eds., D.M., eds.: FME 2003. Volume 2805 of Lecture Notes in Computer Science (Springer-Verlag)., Formal Methods Europe, Springer-Verlag (2003) 94–113
11. Sabatier, D., Lartigue, P.: The use of the B formal method for the design and the validation of the transaction mechanism for smart card applications. In: Proceedings of FM'99: World Congress on Formal Methods. (1999) 348–368
12. Lanet, J.L., Lartigue, P.: The use of formal methods for smartcards, a comparison between B and SDL to model the T=1 protocol. In: Proceedings of International Workshop on Comparing Systems Specification Techniques, Nantes (1998)
13. Casset, L., Burdy, L., Requet, A.: Formal development of an embedded verifier for Java Card byte code. In: DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks, Washington, DC, USA, IEEE Computer Society (2002) 51–58
14. Lano, K.: The B Language and Method: A guide to Practical Formal Development. Springer Verlag London Ltd. (1996)
15. GlobalPlatform: Card Security Requirements Specification. Version 1.0. (2001)
16. Manoranjan, M., Satpathy, M., Butler, M.: ProTest: An automatic test environment for B specifications. In: Proceedings of International workshop on Model Based Testing, ECS University of Southhampton (2004)
17. Ambert, F., Bouquet, F., Chemin, S., Guenaud, S., Legeard, B., Peureux, F., Vacelet, N., Utting, M.: BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. In: Proc. of Formal Approaches to Testing of Software, FATES 2002. (2002) 105–120
18. Stepney, S., Cooper, D., Woodcock, J.: An electronic purse: Specification, refinement and proof. Technical Monograph PRG-126, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK (2000)
19. Cataño, N., Huisman, M.: Formal specification and static checking of Gemplus' electronic purse using ESC/Java. In: FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right, London, UK, Springer-Verlag (2002) 272–289
20. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Software and System Modeling **4** (2005) 32–54
21. Marché, C., Paulin-Mohring, C., Urbain, X.: The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. J. Log. Algebr. Program. **58** (2004) 89–106
22. Burdy, L., Requet, A., Lanet, J.L.: Java applet correctness: A developer-oriented approach. In Araki, K., Gnesi, S., Mandrioli, D., eds.: FME 2003: Formal Methods: International Symposium of Formal Methods Europe. Volume 2805 of LNCS, Springer-Verlag (2003) 422–439
23. VerifiCard project. `http://www.cs.ru.nl/VerifiCard`.
24. Bali project. `http://isabelle.in.tum.de/bali`.

25. Barthe, G., Dufay, G., Jakubiec, L., Serpette, B.P., de Sousa, S.M.: A formal executable semantics of the JavaCard platform. In: ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems, Springer-Verlag (2001) 302–319
26. Alves-Foss, J., ed.: Formal syntax and semantics of Java. Volume 1523 of LNCS. Springer-Verlag (1999)
27. Poll, E., van den Berg, J., Jacobs, B.: Specification of the JavaCard API in JML. In Domingo-Ferrer, J., Chan, D., Watson, A., eds.: Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000), Kluwer Acad. Publ. (2000) 135–154
28. Poll, E., van den Berg, J., Jacobs, B.: Formal specification of the JavaCard API in JML: the APDU class. Computer Networks **36** (2001) 407–421
29. Meijer, H., Poll, E.: Towards a full formal specification of the Java Card API. Volume 2140 of LNCS, Springer-Verlag (2001) 165+
30. Burdy, L., Cheon, Y., Cok, D., Ernst, M.D., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. STTT **7** (2005) 212–232
31. Poll, E., van den Berg, J., Jacobs, B.: Formal specification and verification of JavaCard's application identifier class. In Attali, I., Jensen, T., eds.: Proceedings of the Java Card 2000 Workshop. Volume 2041 of LNCS, Springer-Verlag (2001) 137–150