Model-based Testing with a B Model of the EMV Standard

Roberto Alves de Almeida Junior
Scholarship student of CAPES - Proc. N 8772-11-8
CAPES Foundation, Ministry of Education of Brazil, Brasília/DF
roberto2aj@hotmail.com
s4202872

Supervisor: Prof. Erik Poll

July 3, 2012

ABSTRACT

EMV is a global standard created by Europay, Visa and Mastercard for credit and debit payment cards based on chip card technology. As of 2008, there were more than 730 million EMV-compliant payment cards in use worldwide[EMV12].

The standard is public domain and is described in four books, spanning around 750 pages: [EMV08a][EMV08b][EMV08c] [EMV08d]. The abstraction level of those books is at the level of bytes, making this standard quite hard to understand to the human reader. One way to make it more comprehensible would be through the use of models, however there are no models describing the standard in a high enough level of detail to be used extensively.

In this thesis I present the results of my experiences at trying to create models of the EMV standard using B-Method, a formal method developed by Jean Raymond Abrial and test said models using JTorX.

After providing a background on EMV, B-Method and JTorX, this thesis will describe how to use JTorX to do model-based testing on B models and will also describe the different models created as well as the design decisions behind them.

CONTENTS

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Prof. Erik Poll, who was my research supervisor for this thesis, advising me through this whole semester. I would also like to give special thanks to Prof. Anamaria Martins Moreira for giving me the opportunity to do this research.

I would like to thank both the staff of the Radboud University Nijmegen and the staff of the UFRN (Universidade Federal do Rio Grande do Norte).

I would also like to thank CAPES (Coordenação de Aperfeiçoamento de Pessoas do Nível Superior) for funding my studies here in Nijmegen.

And finally, I would like to thank my family and my friends for their support.

1. INTRODUCTION

This chapter presents an overview of the problem under study, the questions which this research hopes to answer and this chapter will also explain the structure of this thesis.

1.1 Problem Overview

EMV is a global standard created by Europay, Visa and Mastercard for credit and debit payment cards based on chip card technology. As of 2008, there were more than 730 million EMV-compliant payment cards in use worldwide[EMV12].

The standard is public domain and is described in four books, spanning around 750 pages: [EMV08a][EMV08b][EMV08c] [EMV08d]. The abstraction level of those books is at the level of bytes, making this standard quite hard to understand to the human reader. One way to make it more comprehensible would be through the use of models. However there are no models describing the standard in a high enough level of detail to be used extensively.

For my bachelor thesis I plan on making a model of the EMV standard using B-Method, a formal method developed by Jean Raymond Abrial, and try to use this model for model-based testing.

In model based testing, an implementation is tested against a formal model that describes the behavior of the implementation. The tool used for testing will be JTorX.

1.2 Research Questions

While creating and testing the models of the EMV standard, we hope to answer several questions related to B-method, EMV and JTorX.

1.2.1 Research Main Questions

Here are the main questions this thesis hopes to answer. They are the most important questions and will guide the way the research is done as well as tell how successful the research was.

- Is B suitable to make a model of the EMV standard?
- Can we use such a model for model-based testing with JTorX?
- How low in abstraction level can this model be?
- How useful can the model be?
- Can we make a model so low level in abstraction that it can be used for testing with a smart card and/or a terminal?

1.2.2 Sub-questions

Here are the sub-questions this thesis hopes to answer. Those questions are not as important as the main questions, however are questions which we hope to answer during the research.

- Is it possible to make a B model of both the card and the terminal in the EMV standard?
- Given the fact that EMV is a standard that leaves several options open to those who will implement it, is it possible to make 'parametric' models of EMV in B?
- Usually the development of software starts with a description of the software with a high level of abstraction and then as the description is refined more details are inserted into it. However the description we have has a really low level of abstraction, meaning we will have to develop the model in a 'opposite way'. Is B suitable to develop software in this 'opposite way'?
- How can we translate a B model to a format that can be used as input to JTorX?
- JTorX, the tool which will be used for testing, was developed so that not only specialists could use it, but students too. Is this true? Is JTorX useful for both specialists and students or just for specialists?

1.3 Structure

This thesis is organized in the following way. Chapters 2, 3 and 4 will present a background for this thesis, such as concepts related to EMV (Chapter 2), the B-Method (Chapter 3) and JTorX (Chapter 4). Chapter 5 will present how we linked the models in B to JTorX. Chapter 6 will present the first B model. Chapter 7 will present some possible ideas for refinements of the first model. Chapter 8 will present some ideas for future works. Chapter 9 will present some final thoughts regarding the research done. The appendix presents the code in B for the first model of the card and for the first model of the terminal.

2. EMV

This chapter presents a background on the EMV standard, describing its origins, its goals and describing briefly some of its features. More details will be given on Chapter 7 Subsequent Models.

2.1 Origin and Goals

The EMV is a standard for integrated circuit card for payment systems, named after Europay, MasterCard and Visa, the organizations which created this standard in a conjoint effort during mid-90's. EMV has two main objectives:

- Improve security, reducing the possibility of fraud.
- Increase interoperability between banks in different countries so that one card can be used in several countries with no problem of compatibility.

The standard is public, available on `http://www.emvco.com/specifications.aspx`. Today, more than 36 percent of total cards and 65 percent of total terminals deployed are based on the EMV standard[EMV11].

2.2 Specification

The EMV standard is specified in four books [EMV08a], [EMV08b], [EMV08c], and [EMV08d], each book describing different aspects of the standard.

The first book is about the physical and logical characteristics of both the card and the terminal.

The second book is about the security characteristics of the standard.

The third book specifies the structures of the messages traded between the card and the terminal as well as how the general application works.

The fourth book has information about the requirements for devices to be able to communicate with EMC cards.

As of now, the second book was the most relevant for creating the models.

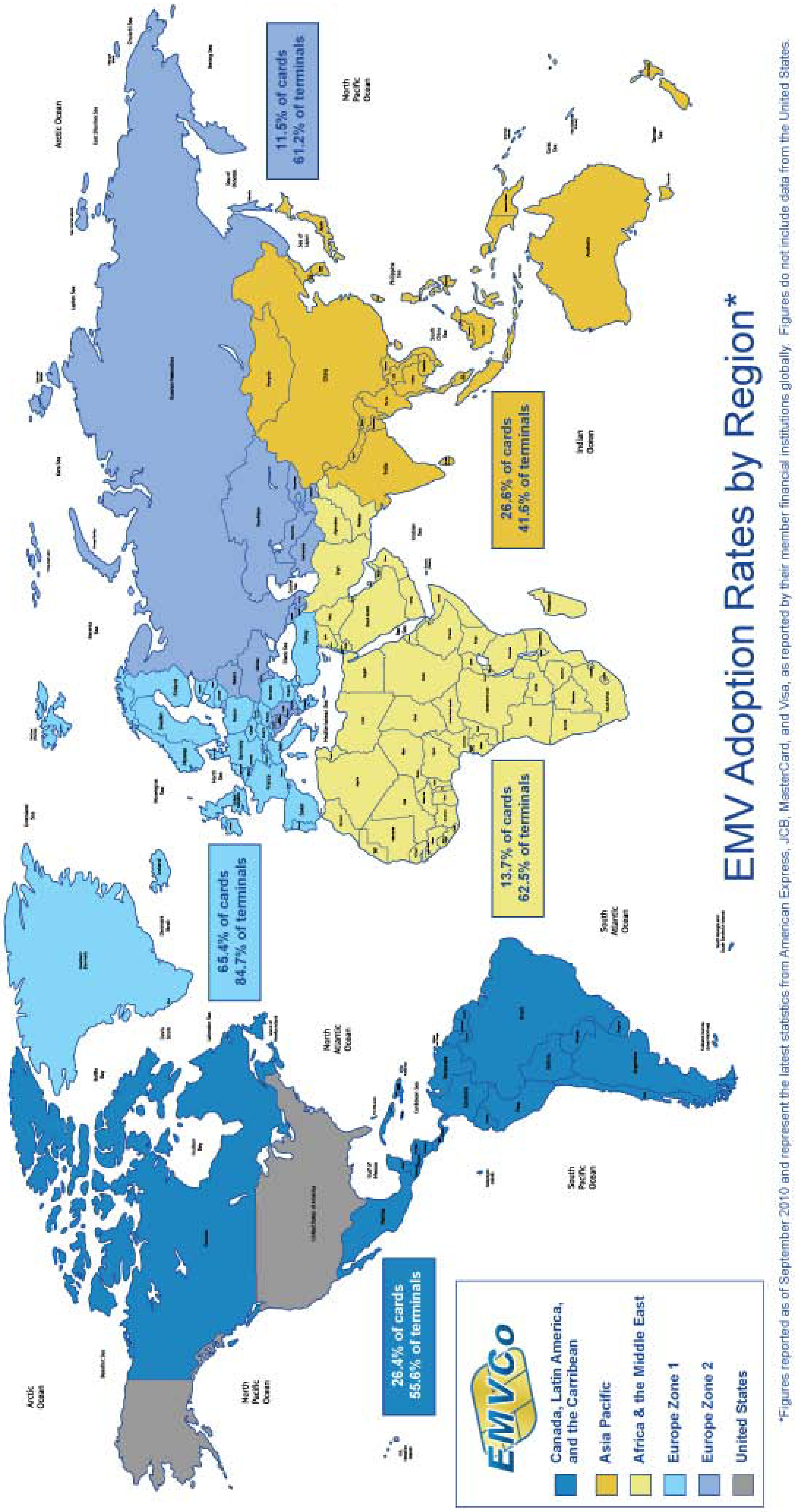

Fig. 2.1: Global EMV Adoption Rates by Region Status September 2010. [EMV11]

3. B METHOD

This chapter presents the B Method, a formal method for software development which was used in this research. First we start with an overview of the B Method, followed by a description of the B Notation and other features of B Method. After that, this chapter will give some background on ProB, the chosen tool for this project.

3.1 B Method Overview

B is a formal method development process language used to specify a software. The objective of B Method is to create highly reliable, portable and maintainable software which correctness can be verified with respect to its functional specification. The system is specified in model of high level of abstraction, then is refined again and again into models with lower level of abstraction until the model can be translated directly into executable code [ALN+91] [Cle12].

3.2 B Notation

This section will describe the basics of B Notation, the language used to specify and model software in B Method.

3.2.1 Basic B Machine Structure

In B the basic building component of a specification is the abstract machine. Here we have the basic structure of an abstract machine in B.

Listing 3.1: B Machine Structure

```
MACHINE ...
VARIABLES ...
INVARIANT ...
INITIALISATION ...
OPERATIONS ...
END
```

The abstract machines in B have 6 important key words:

- Machine - it is the declaration of the machine
- Variables - here we name the variables of the machine. The set of values of these variables define the state of the abstract machine.
- Invariant - here we define properties which must be true independent of the state of the machine.
- Initialisation - here we define the initial values for the variables, thus, the initial state of the machine.
- Operations - here we define the operations of the machine. An operation has pre-conditions and post-conditions. A pre-condition is a condition that must be true for the operation to be used. A post-condition is a condition that must hold true at the end of the execution of the operation.

3.3 ProB

ProB is the chosen tool to work with for making the B models. It is a tool for animation and model checking for B Method that supports automatic refinement between specifications. It was created by Michael Leuschel and Michael Butler [LB08]. It was chosen due to its capability to generate an LTS (Labeled Transition System) directly from the model. ProB uses them to create the animations. The LTS's generated by ProB are in the Graphviz format (.gv, .dot) and are written in dot language, which is a language for graph description.

Initially, another tool, namely AtelierB, was chosen to be used in this research, however this capability to generate LTS automatically proved to be useful as we can use those LTS generated automatically by ProB as an input for JTorX.

4. JTORX

This chapter presents some concepts related to JTorX, a tool for model-driven testing which uses Labeled Transition Systems (LTS) as input. This tool was used to test the B models developed during our research.

4.1 Overview

JTorX is free open-source tool for model-based testing developed by Axel Belifante in 2010 and is available under the BSD license. It can be downloaded at `https://fmt.ewi.utwente.nl/redmine/projects/jtorx/files` .

It is a reimplementation of TorX in Java, making it easier to deploy on different systems. Contrary to TorX, it has a Graphical User Interface (GUI) and is supposedly not only easier to use than TorX but also easier to configure, making it more suitable to be used by students and other non-specialists [Bel10].

As it was said before, JTorX is a tool for model-based testing, which means that it tests the system under study based on a model of the system. For this, JTorX accepts several types of file formats as input, such as Aldebaran (.aut), GraphML (.graphml), GraphViz (.dot,.gv), Jararaca (.jrrc) and Symbolic Transition System (.sax), for example[No 12].

5. LINKING B AND JTORX

This chapter will present how we were able to "link" the B to JTorX so that we were able to run JTorX on the models we created.

As it was mentioned on Chapter 4, ProB is capable of generating animations for B models. For this, it uses the dot tool of the Graphviz package. ProB can export those animations in the Graphviz format (.dot,.gv).

JTorX is able to read files in this format, as long as it can recognize exactly one starting state in the LTS. This is equivalent, in terms of representation in Graphviz, to:

- The graph must have an invisible state(also called node).
- There is a transition from this invisible state to the starting state and said transition cannot have a label.

In case no start state is recognized or multiple starting states are recognized, it will not be possible to use JTorX with the model.

There is another requirement to be able to use JTorX with files in the Graphviz format. The name of the states cannot appear between quotes.

When generating an animation, ProB automatically adds a "Root" node which has a transition from it to the starting state of the model labeled as "initialization". In the file generated all the names of the states appear between quotes, so, in order to use the LTS generated by ProB with JTorX, it is necessary to:

- Turn the "Root" state invisible. This can be done by changing the Style of the node to "invis".
- Take away the label of the "initialization" transition. This can be done by changing the label of the transition to "".
- Take away all the quotes from the names of the states. In this research this was done manually, but for larger state machines it might be useful to develop a program to do that automatically.

This is necessary so that JTorX will recognize the state connected to the Root node as the initial state of the LTS.

6. FIRST MODEL

This chapter presents the first B model made. This chapter describes what part of the standard we modeled as well as some of design choices made. The first model represents only the transaction part of an EMV session and it represents the ADPU's exchanged between the card and the terminal as abstract messages.

6.1 First Model Overview

In this model we have tried to represent a transaction in EMV in the most abstract way possible, so we have abstracted both the card authentication and the cardholder authentication, as seen in Figure 6.1.

To do this, we used as a basis the Figure 6.2, which represents an EMV transaction without the authentication procedures. It is important to note that in the both figures the decision of going either online or offline is represented in a different way. In Figure 6.1, the decision is represented at a higher level of abstraction and happens at just one point. In Figure 6.2, the decision is represented in a lower level of abstraction and can happen at two points. First the terminal has the opportunity to choose between aborting, doing the transaction offline or going online. After that, even if the terminal chose to do the transaction offline, the card can still force the operation to go online. If the terminal decided to go online, the card cannot force the terminal to go offline.

The Figure 6.2 we have a flowchart describing the operation for the system as a whole. For our model we needed instead two flowcharts, one for the card and one for the terminal. We obtained those by breaking the initial flowchart manually into two, as shown in Fig 6.3 and Fig 6.4.

In the Figures 6.3 and 6.4, we represented the card and the terminal in a state diagram similar to UML. For each transition, the figure represents the input that the card/terminal receives. Some transitions are labeled in a different way, though. They can show the output that happens during the transition, which is the case of transition of the card from the state *AACRequested* to *Finished Aborted*. $\_/AAC$ means that the card receives no input and sends an AAC as output.

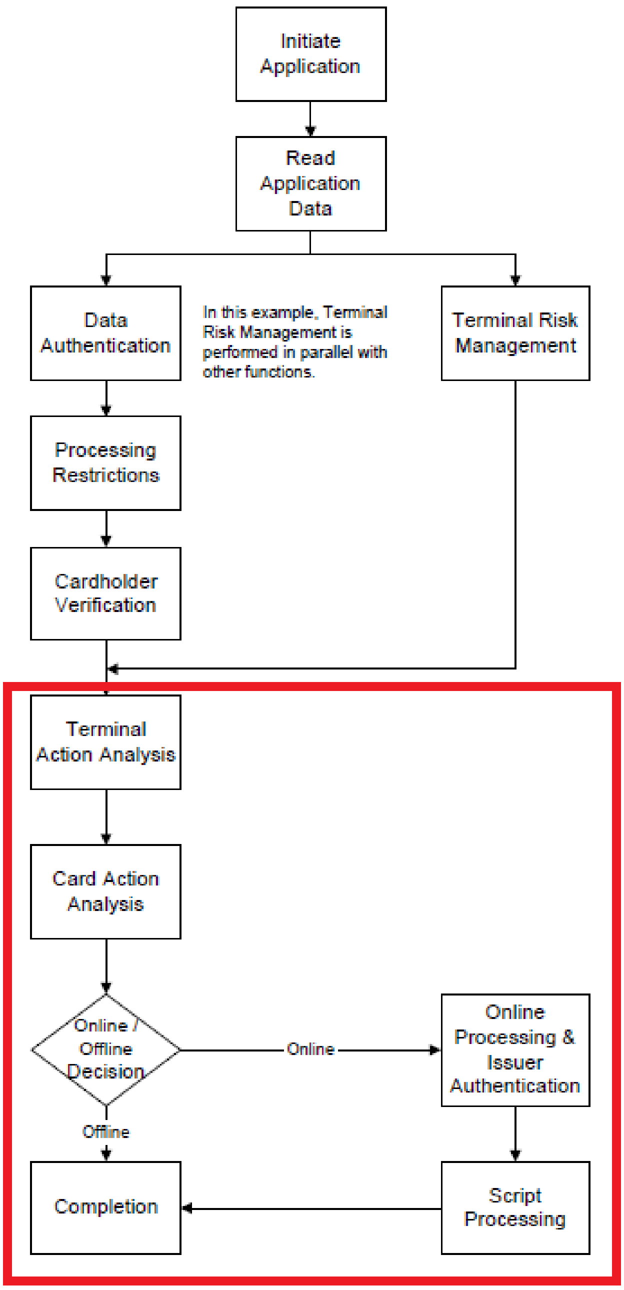

Fig. 6.1: Flowchart describing an EMV transaction. The marked part is the part of the transaction being modeled in the first model.[EMV08c]

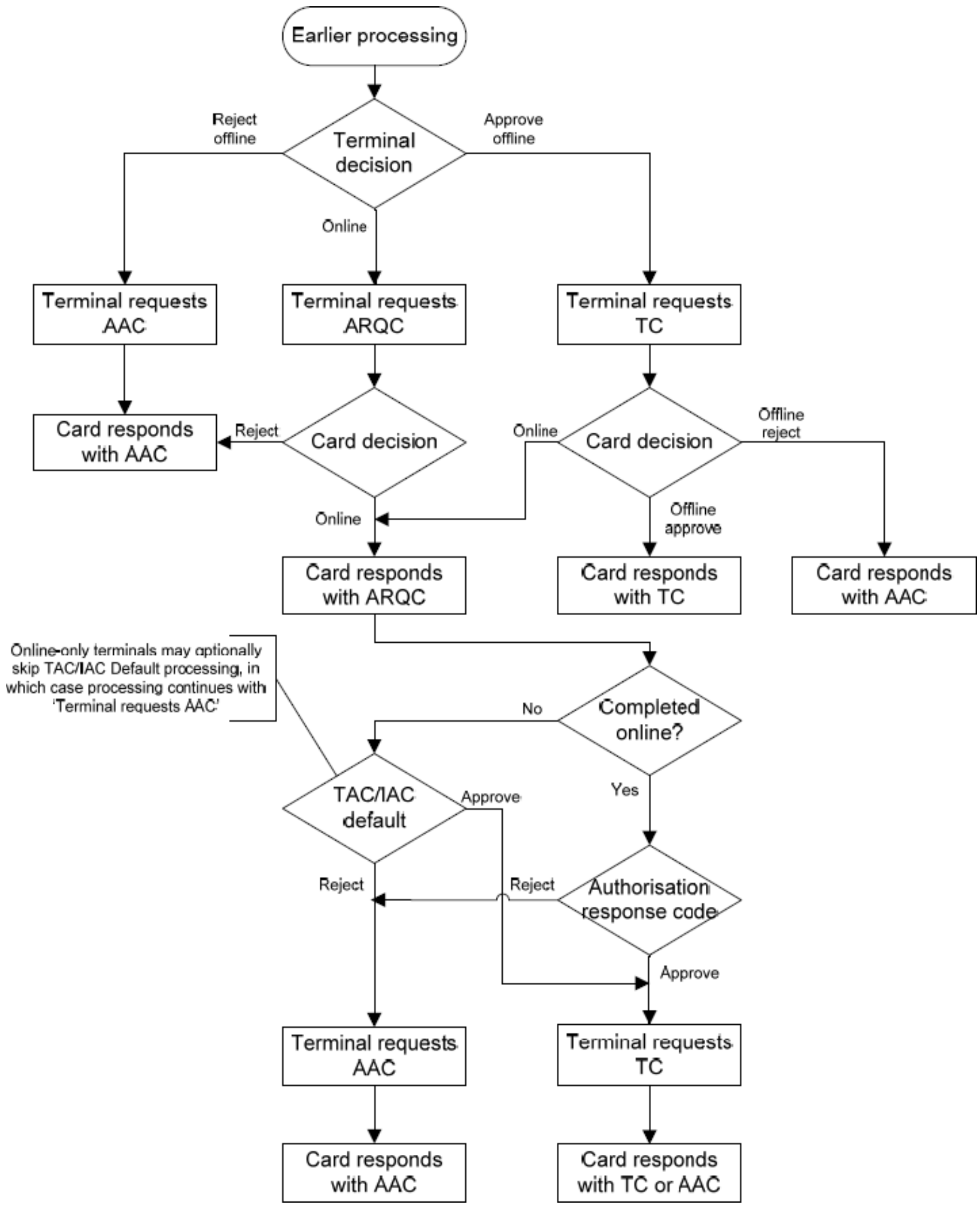

Fig. 6.2: Flowchart describing in detail an EMV transaction without the authentication procedures. [EMV08c]

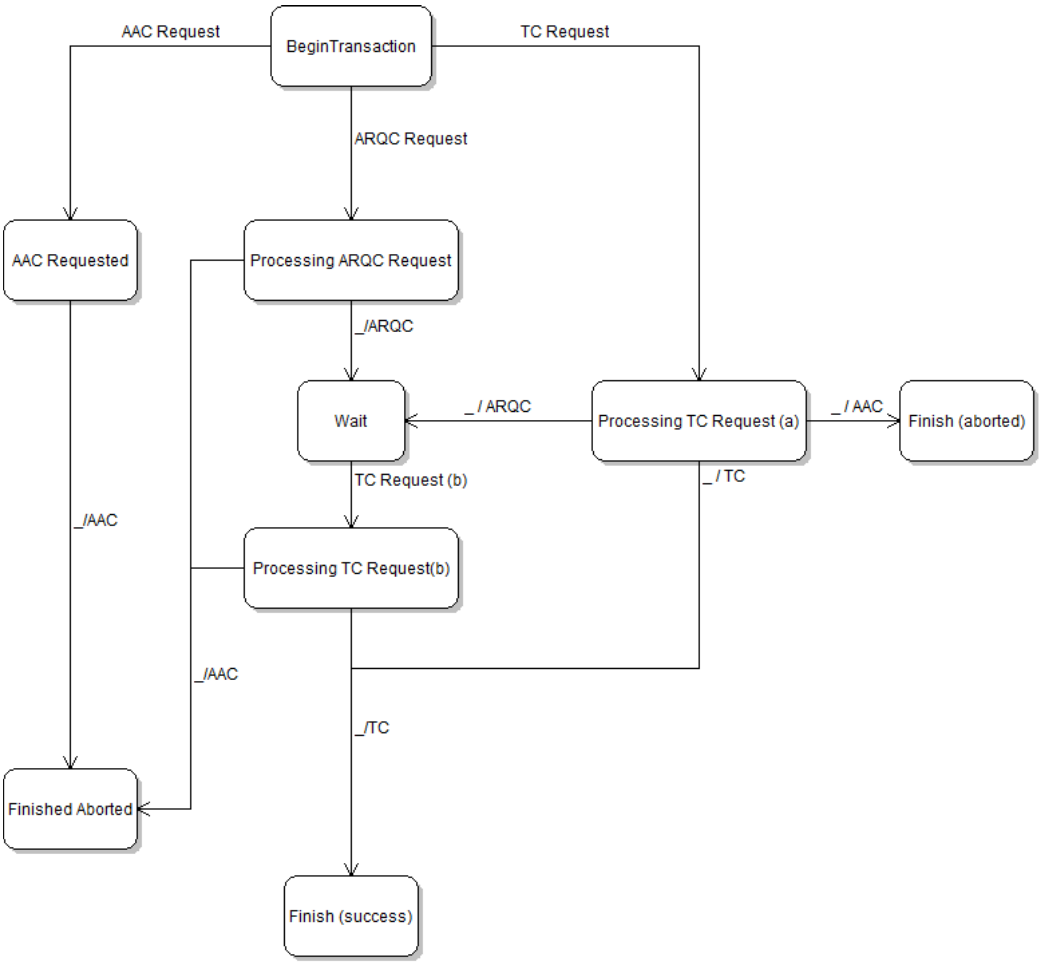

Fig. 6.3: Card flowchart for an EMV transaction.

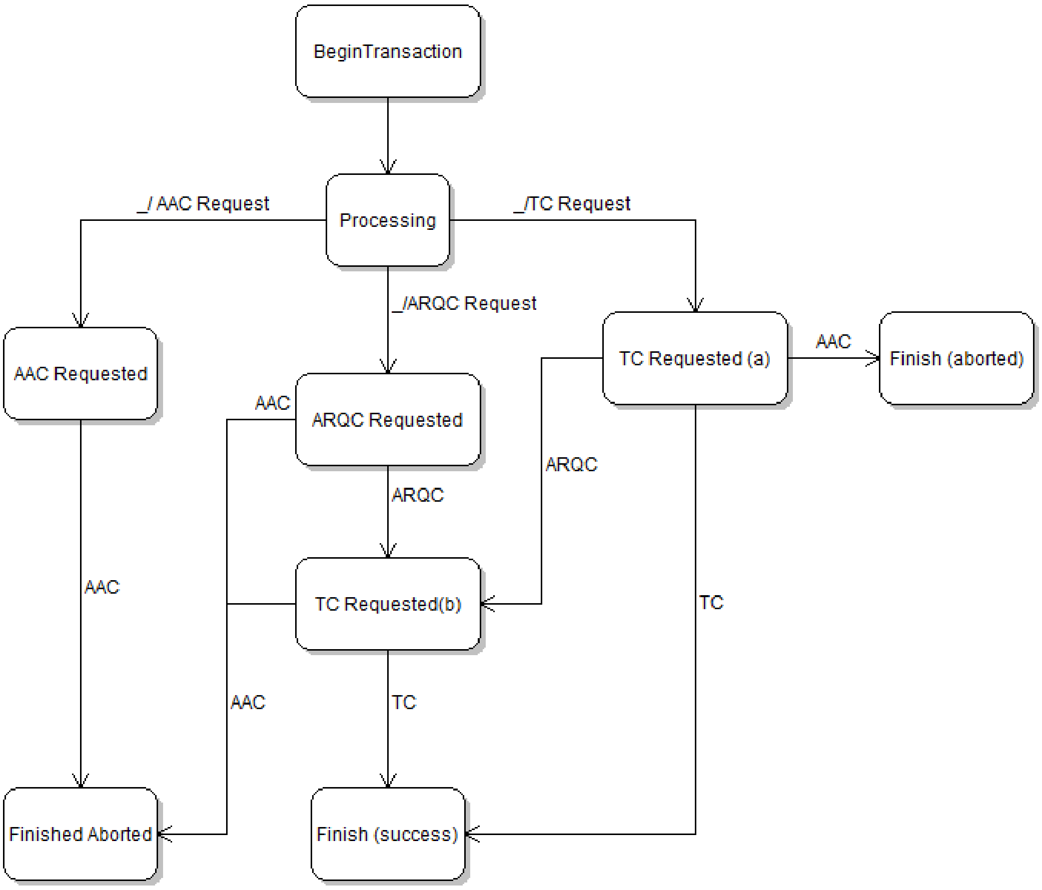

Fig. 6.4: Terminal flowchart for an EMV transaction.

In our model we have abstracted most of the data contained in the messages exchanged between the card and the terminal. In this first model, the only possible difference between two messages is the type. The messages we modeled could be of one of these main types: AC and AC Request. Every message sent from the card to the terminal is represented as an element of the AC set and every message sent from the terminal to the card is represented as and AC Request.

In our model, an AC can be of one of the following types:

- AAC request
- ARQC request
- TC request

In our model, an AC request can be of one of the following types.

- AAC
- ARQC
- TC

Even though they are quite similar and could be represented using the same type, that would be semantically incorrect as the meaning of AAC request is different of the AAC itself. The meaning will become more obvious as the next levels of abstraction are explored.

7. SUBSEQUENT MODELS

This chapter presents some observations on subsequent models, choices made as well as options for future models.

In the first model, as it was seen in the previous chapter, the messages exchanged between the card and the terminal were treated as abstract messages in which only the type of the request made by the terminal of the type of the AC in the response of the card was important.

In the second model, we tried expose the structure of the APDU, except for the data field i.e. we model the class byte, the instruction byte, the p1 byte, the p2 byte, the lc byte and the le byte, though the last three (p2, lc, le) are not used.

In the third model, we tried to expose the structure of the data field i.e. implement TLV data objects in the model. It is important to note that here a lot of fields which did not need implementation at this stage and so they are not being used. Most of the TLV was implemented in this model.

In the fourth model, we hope to take away most of the non-deterministic decisions that we had in the previous models and implement them. In the previous models, as they lacked enough detail to take decisions during some operations, the decision was left non-deterministic. The objective of the model is to change that.

In future models, we hope to expand the model to include other parts of the session besides the transaction and to change the structures used to represent the APDU's as arrays of bytes, which would be the most concrete way of representation.

It is important to note that the fields which are not being used have their value hard-coded. This was done because otherwise the LTS generated by ProB would create one state for every possible value of said field, generating a combinatorial explosion.

It is also important to note that during the research only the first three models were implemented.

Model	Messages
First	Fully abstract. Only the AC type is represented.
Second	Most fields are concrete, except for the data field.
Third	R-APDU data field is now concrete.
Fourth	C-APDU data field is concrete and behavior is also fully implemented.

Tab. 7.1: Comparison between the different models.

7.1 Second Model - Exposing the Structure of the APDU's

In the first model, the only APDU's modeled were those sent/received during a transaction and they were represented in the most abstract possible way. All the content of the APDU was abstracted, except for the type, which in our model was basically the only possible difference between two APDU's. The APDU was represented as two sets, one set for the command APDU's and one set for the response APDU's. An APDU, however, is more complex than just a type, as Figure 7.1 and 7.2 can show.

CLA	INS	P1	P2	Lc	Data	Le
← Mandatory Header[2] →				← Conditional Body →		

Fig. 7.1: Format of the Command APDU. [EMV08c]

For the command APDU we have a mandatory header and a conditional body. The class byte (CLA) and instruction byte (INS) define together the instruction that is being sent to the card. The P1 byte and P2 byte are the parameters. Lc is a byte representing the length in bytes of the data field. Le is a byte representing the expected length of the response.

Data	SW1	SW2
← Body →	← Trailer →	

Fig. 7.2: Format of the Response APDU. [EMV08c]

For the response APDU, besides the data, we have two bytes, SW1 and SW2, called *status word* bytes. They denote the processing state of the command as illustrated in Figure 7.3.

The question now is about the most elegant way of representing this in B. We have came up with the following options:

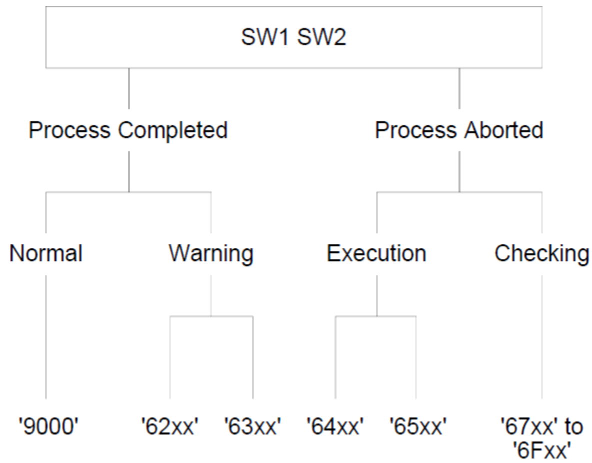

Fig. 7.3: Meaning of the SW bytes. [EMV08c]

- The first option would be to represent the APDU as an array of bytes (the B equivalent of byte). This way of representing the APDU is the closest to its implementation and probably would be the easiest to implement. It is basically a 1:1 conversion from the specification in the book to the code. The major problem of this specification is readability. Once the code is done, it could become really hard to understand it unless the reader had a very good knowledge of EMV. It is too early to use a representation with such low level of abstraction. This option might be useful in a final model, though.

- The second option would be to 'break' the APDU and represent each part separately so that we have direct access to each part of the APDU. This way it would be easier to write preconditions and post-conditions for each operation and the code for the preconditions and post-conditions would be less cluttered too. One of the problems with this options is the fact that the signature for the operations would become big, as each operation would have several parameters. The major problem of this option is the impossibility to return APDU's, since the data is not grouped. Due to this impossibility, this option is not viable at all.

- The third option would be to represent the APDU in a data structure (the B equivalent of a data structure). This way the operations would have only one parameter and the return of data becomes possible too. The only problem is the fact that the code might become cluttered while writing preconditions for the operations as we do not have direct access to each individual part of the APDU, which might make the code hard to read for those who are not familiar with B.

In the end, we opted for the second option for this version of the model. The structure of the APDU was represented in the following way in the code:

Listing 7.1: Example code from the second model of the card.

```
  CLAS_INS = {
    APPLICATION_BLOCK,
    APPLICATION_UNBLOCK,
    CARD_BLOCK,
    EXTERNAL_AUTHENTICATE,
    GENERATE_APPLICATION_CRYPTOGRAM,
    GET_CHALLENGE,
    GET_DATA,
    GET_PROCESSING_OPTIONS,
    INTERNAL_AUTHENTICATE,
    PIN_CHANGE_OR_UNBLOCK,
    READ_RECORD,
    SELECT_,
    VERIFY
  };

  DATA_R = {
    ARQC,
    TC,
    AAC
  };

  REQUEST_TYPE = {
    AAC_REQUEST,
    TC_REQUEST,
    ARQC_REQUEST,
    NO_REQUEST
  };

  DATA = {data_c} /* Abstract type */

(...)

OPERATIONS

  execute(request) =
  PRE
      request : struct(
      clas_ins : CLAS_INS,
          par1 : REQUEST_TYPE,
          par2 : BYTE,
          lc : BYTE,
          data: DATA,
          le :BYTE)
```

```
        & request
                    lc = 0
        & request

                clas_ins = GENERATE_APPLICATION_CRYPTOGRAM
    & request

```

As it can be seen in Listing 7.1, the class and instruction bytes were represented as being one using the set CLAS_INS. This happens because the meaning of the instruction byte depends on the class byte.

Also, the data in the response APDU is represented as being of the type of the AC sent during the session. However it holds much more information than that. The data in the command APDU is represented as an abstract type in this model. The information held in the data field of the APDU's is going to be exposed in the next models.

As of now, the P2 byte in the command APDU and the SW1 and SW2 bytes in the response APDU are not being used. They will be properly implemented together with a proper implementation of the decision making process in the session.

Right now the P1 byte is being represented as a set of possible AC request types, P1 has this meaning only when the command issued by the terminal is the Generate AC. This means the scope of the model starts to include other parts of the session besides the transaction, it will be necessary to represent P1 in another way.

7.1.1 Inconsistency

While analyzing our model one inconsistency was found. As all the results of operations in B must be the same type, it was necessary to create a "dummy AC" for situations in which it can either send a command to request the AC or finish the program, either successfully or by aborting it. One of the operations can be seen in Listing 7.2.

Listing 7.2: Example code from the second model of the terminal showing inconsistency.

```
result <-- TCRequested1(cryptogram) =
PRE
    cryptogram  : R_APDU
  & state       = TC_REQUESTED_1
THEN
    IF    cryptogram.data = AAC  THEN state, result :=
       FINISH_ABORTED,
```

```
            rec(clas_ins: GENERATE_APPLICATION_CRYPTOGRAM,
               p1:NO_REQUEST, p2:0, lc:0, data:DATA, le:0)
        ELSIF cryptogram.data = TC   THEN state, result :=
           FINISH_SUCCESS,
            rec(clas_ins: GENERATE_APPLICATION_CRYPTOGRAM,
               p1:NO_REQUEST,  p2:0, lc:0, data:DATA, le:0)
      ELSIF cryptogram.data = ARQC THEN state, result :=
         TC_REQUESTED_2,
        rec(clas_ins: GENERATE_APPLICATION_CRYPTOGRAM, p1:
           TC_REQUEST,  p2:0, lc:0, data:DATA, le:0)
        END
    END;
```

In the first and second options of the if-then-else the terminal sends an empty request. In the real system no request is sent. However, as the transaction finishes as this message is sent, it might not represent an issue.

7.2 Third Model - Exposing the Structure of the Data Field of the Card

In the second model we exposed the structure of the APDU except for the data field. In this mode we are going the expose the structure of the data field both in the Command APDU and in the Response APDU. However, in the moment of the creation of the third model, the data field of the Response APDU is more important for us as it is where the type of the AC is defined during a transaction whereas the type of the request is defined in the byte P1 in the Command APDU.

The data field of the R-APDU is a BER-TLV data object. There are two types of BER-TLV data objects: primitive and constructed. The primitive BER-TLV is the simplest type, being made of 3 fields: tag (T), length(L) and value(V). The constructed BER-TLV is similar to the primitive one, having the same basic structure, except that its value field is composed of one or more BER-TLV data objects.

Tag (T)	Length (L)	Value (V)

Fig. 7.4: Structure of a primitive BER-TLV data object[EMV08c].

It should be noted that, as for the Response APDU for the Generate AC command, only the primitive BER-TLV is needed, which means that in this model we implemented only the primitive BER-TLV. If the card responds with a TC or an ARQC, the response will contain at least the mandatory

Tag (T)	Length (L)	Primitive or constructed BER-TLV data object number 1	...	Primitive or constructed BER-TLV data object number n

Fig. 7.5: Structure of a constructed BER-TLV data object[EMV08c].

data elements specified in Tab 7.2. The information about the cryptogram sent being a TC or an ARQC is inside of the Signed Dynamic Data.

Tag	Length	Value	Presence
'9F27'	1	Cryptogram Information Data	Mandatory
'9F36'	2	Application Transaction Counter	Mandatory
'9F4B'	N_{IC}	Signed Dynamic Application Data	Mandatory
'9F10'	Var. up to 32	Issuer Application Data	Optional

Tab. 7.2: Data Objects Included in Response to GENERATE AC for TC or ARQC [EMV08b].

If the ICC responds with an ACC, the response will contain at least the mandatory data elements specified in Tab 7.3 instead.

The Listing 7.3 shows how the structure of the TLV was represented for both the C-APDU and the R-APDU.

Listing 7.3: Example code with the structure of the TLV data object.

```
/* Set TLV_TAG, it represents a tag in a tlv data object.
    */
  TLV_TAG = {
      CID_T,
      ATC_T,
      SDAD_T,
      AAC_T,
      IAD_T
  };

```

Tag	Length	Value	Presence
'9F27'	1	Cryptogram Information Data	Mandatory
'9F36'	2	Application Transaction Counter	Mandatory
'9F26'	8	AAC	Mandatory
'9F10'	Var. up to 32	Issuer Application Data	Optional

Tab. 7.3: Data Objects Included in Response to GENERATE AC for AAC [EMV08b].

```
/* structure of the response APDU */
  response : struct(
    sw1  : BYTE,
    sw2  : BYTE,
    data : struct (
      cid         : struct (tag : TLV_TAG, length : NAT,
          value : NAT),
      atc         : struct (tag : TLV_TAG, length : NAT,
          value : NAT),
      sdad_or_aac : struct (tag : TLV_TAG, length : NAT,
          value : AC_TYPE),
      iad         : struct (tag : TLV_TAG, length : NAT,
          value : NAT, null : BOOL)
    )
  )
```

As the field for the IAD (Issuer Application Data) is optional, we added a boolean to determine whether it is being used or not.

As it can be seen in the Listing 7.3, there is a TLV called ”ssad_or_aac”. It was created in order to have uniformity what is being sent, be it an AAC, an ARQC or a TC. This structure represents the TLV in the third row of Tab 7.2 and Tab 7.2, depending on the value of the tag being used. When the cryptogram being sent is an AAC, the value of the tag ”ssad_or_aac” is AAC_T (AAC Tag) and the value of the value field is ”AAC”, otherwise the value of the tag is SSAD_T (Signed Dynamic Authentication Data Tag) and the value of the value field is equal to the name of the cryptogram type being sent.

As of now, only the following fields are being used in the model:

- The tags.
- The length fields, except for the SDAD.
- The value referred by the ”ssad_or_aac” tag.

The SSAD, however, holds more data than that in its value field. Such data will be uncovered in the next models.

8. FUTURE WORKS

This chapter presents the idea of possible future works based on the research that has been done while doing this thesis.

This work has paved the way on how to make a B model of the EMV standard and how to test such models using JTorX. A possible continuation of this work is to refine this model, lowering the abstraction level even more, adding behavior and adding other parts of the EMV session besides the transaction.

9. CONCLUSIONS

In the beginning I had big ambitions for this project, as I thought it would be possible to model the whole standard in high detail and, at the end, have several models at disposal, however the time was really short for me to be able to do such a thing. Instead, I think I could pave the way so that such objectives might be attained in future projects.

I did not find JTorX an easy tool to use. Though the tool does have a clean interface, the lack of a manual makes it hard to use, unless the user has either prior knowledge about the theory behind the tool or has someone to teach him/her how to use the tool. JTorX might be useful in teaching about model-driven test, but it seems to not be very useful for beginners who are trying to use the tool without any help.

As for the design choices during the creation of the models, I believe that I would not start by modeling only the transaction if I was more focused in testing in the beginning. The main focus was to create models

BIBLIOGRAPHY

[ALN$^+$91] J. Abrial, M. Lee, D. Neilson, P. Scharbach, and I. Srensen. The B-method. In Sren Prehn and Hans Toetenel, editors, *VDM '91 Formal Software Development Methods*, volume 552 of *Lecture Notes in Computer Science*, pages 398–405. Springer Berlin / Heidelberg, 1991.

[Bel10] Axel Belinfante. JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer Berlin / Heidelberg, 2010.

[Cle12] ClearSy. Presentation of the B Method. http://exoplanet.eu/catalog.php, accessed in February 2012.

[EMV08a] EMVCo. EMV– Integrated Circuit Card Specifications for Payment Systems, Book 1: Application Independent ICC to Terminal Interface Requirements, 2008.

[EMV08b] EMVCo. EMV– Integrated Circuit Card Specifications for Payment Systems, Book 2: Security and Key Management, 2008.

[EMV08c] EMVCo. EMV– Integrated Circuit Card Specifications for Payment Systems, Book 3: Application Specification, 2008.

[EMV08d] EMVCo. EMV– Integrated Circuit Card Specifications for Payment Systems, Book 4: Cardholder, Attendant, and Acquirer Interface Requirements, 2008.

[EMV11] EMVCo. A Guide to EMV, 2011.

[EMV12] EMVCo, LLC. About EMV. `http://www.emvco.com/about_emv.aspx`, accessed in February 2012.

[LB08] Michael Leuschel and Michael Butler. ProB : an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer (STTT)*, 10:185–203, 2008.

[No 12] No author given. JTorX - a tool for Model-Based Testing. https://fmt.ewi.utwente.nl/redmine/projects/jtorx/wiki/, accessed in February 2012.

APPENDIX

A. B CODE OF THE FIRST MODEL

Listing A.1: Code of the Card

```
/* CardV1
 * Author: Roberto
 * Creation date: dom mar 18 2012
 */
MACHINE
  CardV1
SETS
  CARD_STATE = {
    BEGIN_TRANSACTION,
    PROCESSING_AAC_REQUEST,
    PROCESSING_ARQC_REQUEST,
    PROCESSING_TC_REQUEST_1,
    PROCESSING_TC_REQUEST_2,
    WAIT,
    FINISH_ABORTED,
    FINISH_SUCCESS
  };

  AC = {
    ARQC,
    TC,
    AAC
  };

  REQUEST = {
    AAC_REQUEST,
    TC_REQUEST,
    ARQC_REQUEST,
    NO_REQUEST
  }

VARIABLES
  state

INVARIANT
  state : CARD_STATE

```

```
INITIALISATION
  state := BEGIN_TRANSACTION

OPERATIONS

  execute(request) =
  PRE
    request     : REQUEST
      & request /= NO_REQUEST
        & state     = BEGIN_TRANSACTION
    THEN
    IF (request = ARQC_REQUEST) THEN
      state := PROCESSING_ARQC_REQUEST
    ELSIF (request =   TC_REQUEST) THEN
        state := PROCESSING_TC_REQUEST_1
    ELSIF (request =  AAC_REQUEST) THEN
          state := PROCESSING_AAC_REQUEST
    END
  END;

  cryptogram <-- processAACRequest =
  PRE
    state      = PROCESSING_AAC_REQUEST
  THEN
      state, cryptogram := FINISH_ABORTED, AAC
  END;

  cryptogram <-- processARQCRequest =
  PRE
    state = PROCESSING_ARQC_REQUEST
  THEN
      ANY response WHERE response : AC & response /= TC
         THEN
          IF response =  AAC THEN
        state, cryptogram := FINISH_ABORTED, response
      ELSIF response = ARQC THEN
          state, cryptogram := WAIT,           response
      END
      END
  END;

  cryptogram <-- processTCRequest_1 =
  PRE
      state      = PROCESSING_TC_REQUEST_1
  THEN
      ANY response WHERE response : AC THEN
      IF    response = TC   THEN
        cryptogram, state := response, FINISH_SUCCESS
      ELSIF response = AAC  THEN
```

```
          cryptogram , state := response , FINISH_ABORTED
        ELSIF response = ARQC THEN
          cryptogram , state := response , WAIT
            END
        END
    END;

      wait(request) =
      PRE
              request : REQUEST
            & request = TC_REQUEST
            & state   = WAIT
    THEN
            state := PROCESSING_TC_REQUEST_2
    END;

    cryptogram <-- processTCRequest_2 =
    PRE
      state    = PROCESSING_TC_REQUEST_2
    THEN
        ANY response WHERE response : AC & response /= ARQC
            THEN
            IF    response =  TC THEN
          cryptogram , state := response , FINISH_SUCCESS
        ELSIF response = AAC THEN
          cryptogram , state := response , FINISH_ABORTED
        END
        END
    END

END
```

B. B CODE OF THE FIRST MODEL

Listing B.1: Code of the Terminal

```
/* TerminalV1
 * Author: Roberto
 * Creation date: dom mar 18 2012
 *
 * This is the first model of the terminal.
 */
MACHINE
    TerminalV1

SETS
    TERMINAL_STATE = {
        BEGIN_TRANSACTION,
        PROCESSING,
    AAC_REQUESTED,
    ARQC_REQUESTED,
    TC_REQUESTED_1,
    TC_REQUESTED_2,
        ONLINE_PROCESSING,
        FINISH_SUCCESS,
        FINISH_ABORTED
    };

    AC = {
        ARQC,
        TC,
        AAC
    };

    REQUEST = {
        AAC_REQUEST,
        TC_REQUEST,
        ARQC_REQUEST,
        NO_REQUEST
    };

  TERMINAL_DECISION = {
        WILL_REQUEST_AAC,
```

```
        WILL_REQUEST_TC,
        WILL_REQUEST_ARQC,
        NO_DECISION
    }

VARIABLES
  state, previous_decision

INVARIANT
     state : TERMINAL_STATE
  & previous_decision : TERMINAL_DECISION

INITIALISATION
  state, previous_decision := BEGIN_TRANSACTION,
     NO_DECISION

OPERATIONS

    /*This operation represents what happens before the
       transaction so that the terminal makes its first
       decision */
    /*It exists so that the value for "previousDecision"
       is not hardcoded*/
    pre_execute =
    PRE
        previous_decision = NO_DECISION
        & state = BEGIN_TRANSACTION
    THEN
        ANY new_decision WHERE new_decision :
           TERMINAL_DECISION & new_decision /=
           NO_DECISION THEN
          state,previous_decision := PROCESSING,
             new_decision
        END
    END;

    result <-- execute =
    PRE
        state = PROCESSING
        & previous_decision /= NO_DECISION
    THEN
        IF    (previous_decision = WILL_REQUEST_AAC )
           THEN state, result :=  AAC_REQUESTED,
           AAC_REQUEST
        ELSIF (previous_decision = WILL_REQUEST_TC   )
           THEN state, result :=   TC_REQUESTED_1,
           TC_REQUEST
    ELSIF (previous_decision = WILL_REQUEST_ARQC) THEN
       state, result := ARQC_REQUESTED,  ARQC_REQUEST
```

```
        END
    END;

    AACRequested(cryptogram) =
  PRE
      cryptogram : AC
    & state      = AAC_REQUESTED
    & cryptogram = AAC
  THEN
    state := FINISH_ABORTED
  END;

  result <-- TCRequested1(cryptogram) =
  PRE
      cryptogram  : AC
    & state       = TC_REQUESTED_1
  THEN
      IF    cryptogram = AAC  THEN state, result :=
          FINISH_ABORTED, NO_REQUEST
    ELSIF cryptogram = TC   THEN state, result :=
        FINISH_SUCCESS, NO_REQUEST
      ELSIF cryptogram = ARQC THEN state, result :=
          TC_REQUESTED_2, TC_REQUEST
      END
  END;

  result <-- ARQCRequested(cryptogram) =
  PRE
       cryptogram  : AC
    &  state       = ARQC_REQUESTED
    & (cryptogram = AAC or cryptogram = ARQC)
  THEN
      IF    cryptogram = AAC  THEN state, result :=
          FINISH_ABORTED, NO_REQUEST
      ELSIF cryptogram = ARQC THEN state, result :=
          TC_REQUESTED_2, TC_REQUEST
      END
  END;

  TCRequested2(cryptogram) =
  PRE
       cryptogram  : AC
    &  state       = TC_REQUESTED_2
      & (cryptogram = AAC or cryptogram = TC)
  THEN
      IF    cryptogram = AAC  THEN state :=
          FINISH_ABORTED
    ELSIF cryptogram = TC   THEN state := FINISH_SUCCESS
    END
```

```
    END

END
```

LIST OF ACRONYMS AND ABBREVIATIONS

AAC	Application Authentication Cryptogram
AMN	Abstract Machine Notation
APDU	Application Protocol Data Unit
ARPC	Authorization Response Cryptogram
ARQC	Authorization Request Cryptogram
ATM	Automated Teller Machine
BER	Basic Encoding Reference
CDA	Combined Data Authentication
CID	Cryptogram Information Data
C-APDU	Command APDU
CDOL	Card Risk Management Data Object List
CLA	Class Byte
CVM	Cardholder Verification Method
DDA	Dynamic Data Authentication
DES	Data Encryption Standard
EMV	Europay, Mastercard and Visa
GUI	Graphical User Interface
ICC	Integrated Circuit Card
INS	Instruction Byte
PIN	Personal Identification Number
POS	Point of Sale
R-APDU	Response APDU
RFU	Reserved for Future Use
RSA	Rivest, Shamir, Adleman Algorithm
SDA	Static Data Authentication
TAL	Terminal Application Layer
TC	Transaction Certificate
TLV	Tag Length Value

LIST OF FIGURES