Is Javacardsign correct and secure? Bachelor Thesis

Robert Kleinpenning rkleinpe@science.ru.nl

July 14, 2012

Supervisors:

dr.ir. E. Poll
E.Poll@cs.ru.nl

dr. W.I. Mostowski
E.Poll@cs.ru.nl

Abstract

"Javacardsign" is a Java Card Project created at the Radboud University Nijmegen by Wojciech Mostowski who is a member of the Digital Security Group. It consists of an on-card applet and host application. The applet can be used for ISO 7816 [1] compliant digital signatures, decryption and authentication. This thesis assesses the correctness of the on-card implementation by means of model based testing. The host application is not considered in this thesis.

This thesis explains how the applet was modelled, and how model based testing was used to validate that the Javacardsign applet is correct and secure. The methodology used in this thesis could be used to test other smart cards that are a component of a Public Key Infrastructure.

1 Introduction

Many EU Member States are considering introducing a citizen digital signature card. In particular the Dutch government is working on a similar project called the Electronic Dutch Identification Card (e-NIK) [2]. This will allow the card holder to reliably use the electronic services provided by the government, using public key cryptography. This might replace the current DigiD project. This thesis will provide a methodology to comprehensively test applications similar to e-NIK.

Javacardsign[3] consists of an on-card applet and a host application. The host can be some terminal with a card reader that communicates with the card. The focus of this thesis is testing of the on-card implementation of the Javacardsign applet. This implementation is written in Java Card and it processes the messages sent to the card, next to managing the file storage for certificates and performing cryptographic operations. The card can sign data, decrypt data, and it can authenticate itself to the terminal. These cryptographic operations are guarded by a PIN.

The Javacardsign applet is used for creating and checking ISO 7816 [1] compliant digital signatures. The ISO 7816 specifications parts 4, 8 and 15 were used to develop this software. These sections cover card commands and cryptographic operations used in digital signature applications.

1.1 Example Scenario

To explain how the Javacardsign applet works, we provide an example of computing a digital signature. For this several messages are used.

- The terminal starts with a Manage Security Environment instruction, which selects and prepares keys for computing the signature. Signing, decrypting and authenticating use different keys. The selection of keys is required prior to any cryptographic operation. So for *signing* the card will prepare the *sign key*.
- Next the terminal sends a Verify instruction which requires a correct PIN. This pin is provided by the user to the terminal. This will allow the user of the card to authenticate. If an incorrect PIN is used, the card replies with an error.
- Finally, the Perform Security Operation instruction is executed, the card will sign the data the terminal provides. If the terminal is not verified, or if there is no key prepared, the card will reply with an error.

It is important to note that each of these instructions are atomic regarding the communication between the terminal and card. They are all single messages. Also, this is the *required* order to successfully compute a digital signature. However, it is still possible to give different commands. This will cause the card to reply with an error and remain in its current state.

1.2 Commands

The following commands are supported by the card, after personalisation.

1. Change Reference Data - Change the PIN with the PUC.
2. Verify - Verify the user with the PIN.
3. Manage Security Environment - Prepare a key for signing, decrypting or authenticating.
4. Perform Security Operation - Sign or decrypt data provided by the terminal.
5. Internal Authentication - Authenticate to the terminal.

2 Messaging

The Javacardsign applet and host application use APDUs to communicate. APDU stands for Application Protocol Data Unit. It is the communication unit between a card terminal and a card. These messages must comply with the specifications defined in the ISO 7816 standard. An APDU consists of multiple bytes, in a particular order.[4] The terminal sends *command APDUs* and the card replies with *response APDUs*. APDUs contain several mandatory bytes. In the case of command APDUs, these represent the instruction of the APDU. Next to the obligatory bytes, there is an optional payload. Response APDUs contain the result of the command in a status word that consists of two bytes. The value of these status words represents the success of a command or, in case of an error, the corresponding error value. The response can also contain a payload.

3 Approach

The testing method used in this thesis is similar to the testing methods by Mostowski et al. [5, 6] on electronic passports. The work done to complete this project consists of the following:

- Phase 1: Exploring the Javacardsign applet.
  By modelling some of the commands in detail, using Uppaal, I was able to understand what the source code was supposed to do. Uppaal was only used as a graphical editor to model the source code of the applet, so most of the features were unused. The models drawn at this stage can be found in the appendix (see section 9). These were not used for testing purposes. The purpose of these models was to assist in understanding the Java Card code.
- Phase 2: Modelling.
  Using the information gained in phase 1, two models were created: Model 1 [figure 1] and Model 2 [figure 3]. This time using yEd (see section 4.1).

These models were used to test the correctness of the on-card implementation. Model 1 was used to check if the card shows correct behaviour, in normal conditions. Model 2 was designed to check for the absence of incorrect behaviour. These models are more abstract then the ones created in phase 1, because the application can only be tested on a *per-APDU level*. It is only possible to check the result the card provides, not how the card calculated this result.

- Phase 3: Model based testing.
  The testing was done using JTorX (section 5.2). A smart card reader and a card with the implementation were also used for the tests. Model based testing was used to examine the implementation. After testing the results are analysed, to see if the Javacardsign applet is correct and secure.

Based on these findings a test report is given, providing the conclusion whether or not Javacardsign is correct and secure with respect to the formal models I made.

4 Modelling

Two main models were designed for use in phase 3 [section 5]. These models are referred to as Model 1 [figure 1)] and Model 2 [figure 3]. Model 1 only checks for correct behaviour, whereas Model 2 also checks for absence of incorrect behaviour.

4.1 yEd

yEd is a diagram editor that can be used to generate models that JTorX can interpret as input. It was used to create Model 1 and Model 2. The models use a very simple syntax. Commands sent to the smart card end with a question mark, responses are shown with exclamation marks.

4.2 Model 1

Model 1 was designed to see if the applet was working as intended. It only modelled correct behaviour as can be seen in figure 1. In the following models two main colours are used. The yellow (light) states represent input states, and the blue (dark) states represent output states. In the yellow states the terminal is required to do an action, and the blue states require an action of the smart card. Because we are modelling from the viewpoint of the terminal, all blue states have an "!" toward a yellow state (response) and all yellow states have a "?" to the blue states (command).

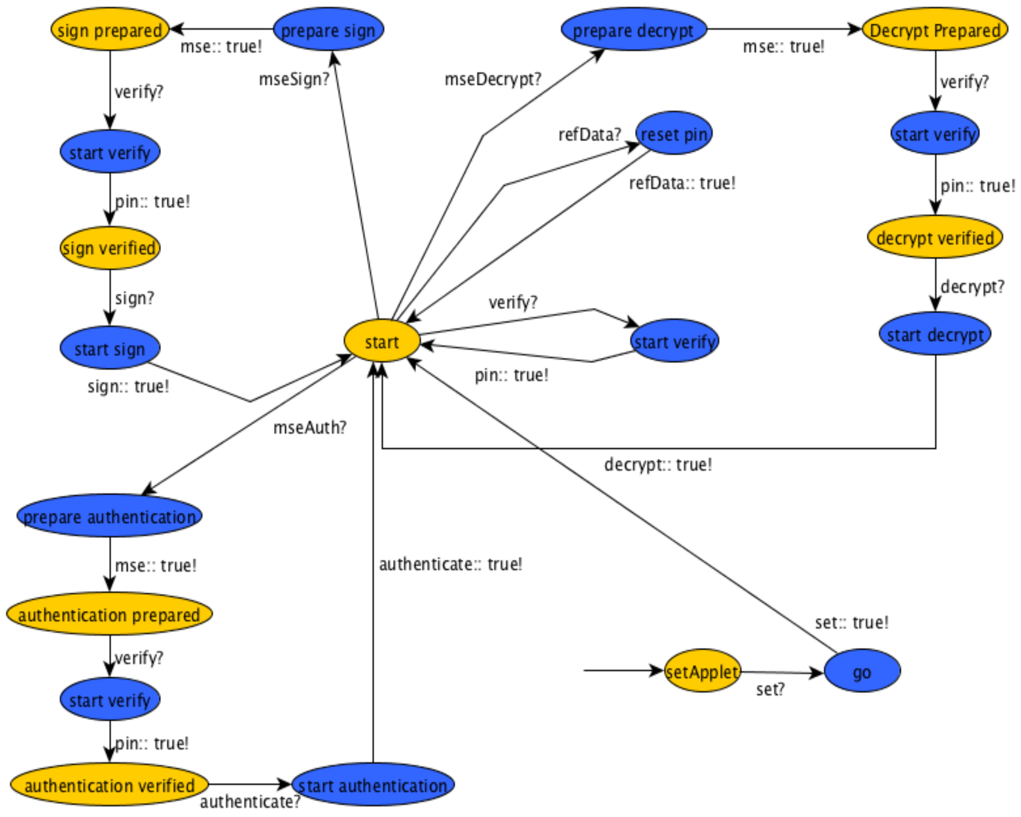

Figure 1: Model 1 - Modelling correct behaviour

In this model three main branches can be identified, and two small ones. The two small ones have only one state each: reset: pin and start verify. These are legal actions, and should be allowed at the start state. The three main branches model the core functionality of the Javacardsign applet, specifically signing, decrypting and authenticating. All three start with a similar command: mseSign, mseDecrypt and mseAuth. "mse" Stands for Manage Security Environment, and these commands prepare the key for the security sensitive action (start sign, start decrypt or start authenticate) further in the branch. The card should reply with a confirmation that the key is prepared. Before the security sensitive operation can be performed, the card holder must be verified. This is done with the PIN in the start verify step. After verification the card performs the security operation, and replies with the result, the model returns to the start state. The content of the results is checked by the adapter.

After authentication the model returns to the start state. However, this is an abstraction. After authentication without entering the PIN and while no other key is prepared, it is possible to authenticate again. This was not modeled because it would have little to no effect on the tests, and it is considered out of the scope of this paper.

4.3 Model 2

The second model checks for the absence of incorrect behaviour. This uses an extended version of Model 1. With this model JTorX tests specific behaviour that should not be accepted by the card.

For example, if we look at state decrypt prepared in Model 1, the card should not allow an attempt to sign something while the decryption key is prepared by the Manage Security Environment command. If the card gives a response that indicates that it accepted the sign command, JTorX will end up in an error state, halt and return the *failed* verdict.

To force JTorX to try these options, they are added manually. The extra behaviour that is tested in Model 2 is limited to signing, decrypting and authenticating. Only these commands are considered security sensitive. By using the extension shown in figure 2 JTorX can attempt to find ways to sign, decrypt or authenticate in states where this should not be possible. The verify, change reference data, and different manage security environment commands are omitted in this extension. The reason for this is because these operations do not require PIN verification, and are therefore less relevant for the correctness and security of the Javacardsign applet. Attempts to sign, decrypt and authenticate are done at every input state.

The model has three *security sensitive states* meaning the states where the card can perform the security operations. These states are sign verified, decrypt verified and authentication verified. In these states one of the security sensitive commands is already present. Using the same extension on these state would result in a false negatives. So in these particular cases the correct command is omitted from the extension.

If we look at figure 2 we see a random input state at the bottom and an otherwise state. JTorX will randomly enter the otherwise state by trying one of the security operations. Once in this otherwise state, JTorX waits for the response from the card. The response should be false in all cases, because the card should not execute these commands. If the response is anything else, the model will halt and the test will fail.

For this, several states, labeled otherwise, have been added to Model 1. Model 2 is Model 1 combined with the extension. However, for readability, the extension has been reduced to one edge, namely otherwise? as edge and a red (dark) state called error state.

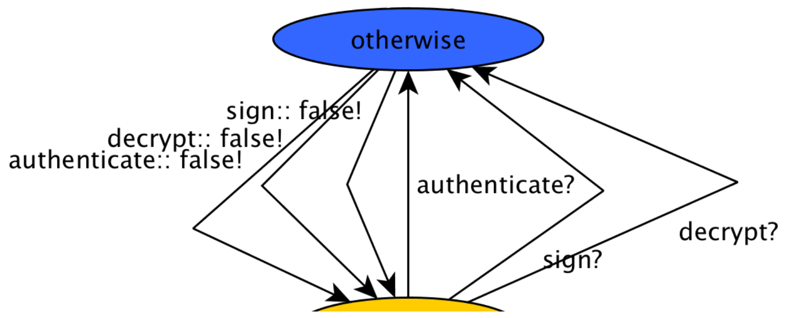

Figure 2: An otherwise state

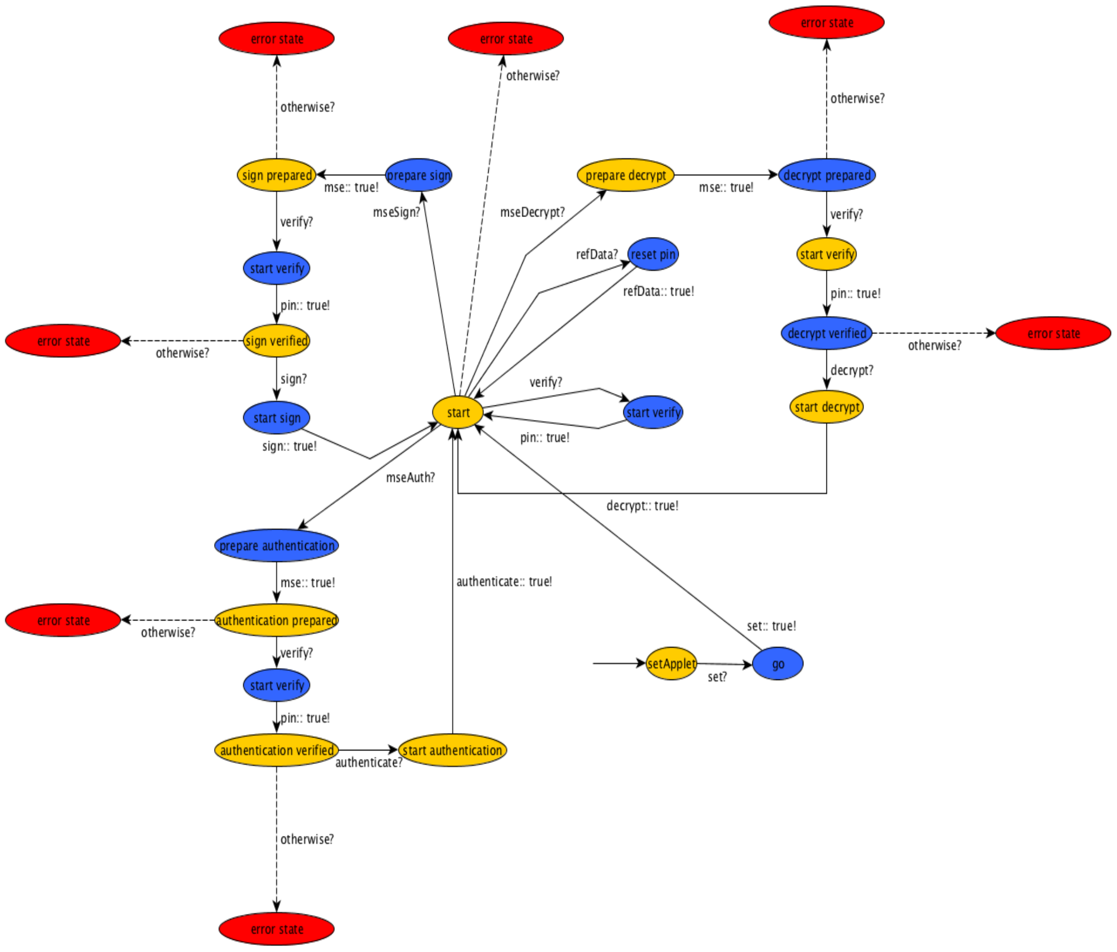

Figure 3: Model 2 - Modelling absence of incorrect behaviour

5 Model based testing

5.1 Setup

For testing a basic setup was used. The smart card communicates with an adapter script running on the laptop. JTorX also communicates with the adapter. The adapter was created to translate commands JTorX bases on the model it reads, to APDUs the smart card can understand, and vice versa.

Once communication was setup up correctly, simple models were tested to explore the tool. Once these worked as intended, Model 1 and Model 2 were tested.

5.2 JTorX

JTorX is a tool developed by Axel Belinfante.[7] It is a tool for model based testing. JTorX makes random steps in the model, it sends commands to the smart card and verifies the responses. This is done by comparing the response with the response that was expected by the model.

JTorX is a model based testing tool. It supports two formats, Aldebaran (.aut) or GraphML (.graphml) format. yEd creates models in the latter format. JTorX was used to read Model 1 and Model 2. JTorX randomly tries the different options these models provides on the system being tested. JTorX is able to interact with programs on their standard input and output, like the adapter (section 5.3). These programs are actually what is being tested.

JTorX generates three models during testing. One model is an interpretation of the model created in yEd, because yEd works well with JTorX, this is an exact copy. The second one is a message sequence chart. This is a visualised log of the test run. The third model is a finite state model, which is constructed based on the actions JTorX performed.

5.3 Adapter

The adapter handles communication between JTorX and the smart card. JTorX sends commands it reads in the model to the adapter. The adapter then sends the corresponding APDU to the smart card, and reports back the result. JTorX only sends simple commands to the adapter.

The adapter is a command line application that accepts the input that is shown in the models above. For example, JTorX will send mseSign? to the adapter. The adapter then translates this to the corresponding APDU, and sends it to the smart card. The smart card responds with a response APDU. In the case of mseSign, the response APDU only consists of a *status word*. If the command is successful, this status word will be 9000. In other cases it will have a value that corresponds with a certain error. For this paper it is not relevant what exactly went wrong in case of an error. So the adapter only checks if the status word is 9000, and returns *mse:: true!* or *mse:: false!* accordingly to JTorX.

For the more complex commands: *sign*, *decrypt* and *authenticate*, the adapter also checks if the payload the card returns is equal to the expected data. In the case of decrypt the adapter sends two chained APDUs that contains the encrypted text "hallo". The smart card decrypts this text, and returns the result in a response APDU. The adapter then checks both the status word, and the result. If both are correct, then it will return decrypt:: true! to JTorX, otherwise it will return decrypt:: false!. Sign and authenticate are handled in similar ways, the adapter handles all communication, and simply returns true or false.

6 Results of the tests

Figure 4 shows a short extract of the message sequence diagram. This diagram shows the choices JTorX made during the test using Model 2. It is a visualised log of the test run. Something that JTorX adds is a delta transition. This transition occurs when JTorX receives no response from the adapter, because the adapter is still waiting for the cards response. When this happens, JTorX uses a provided timeout to wait, and to try again.

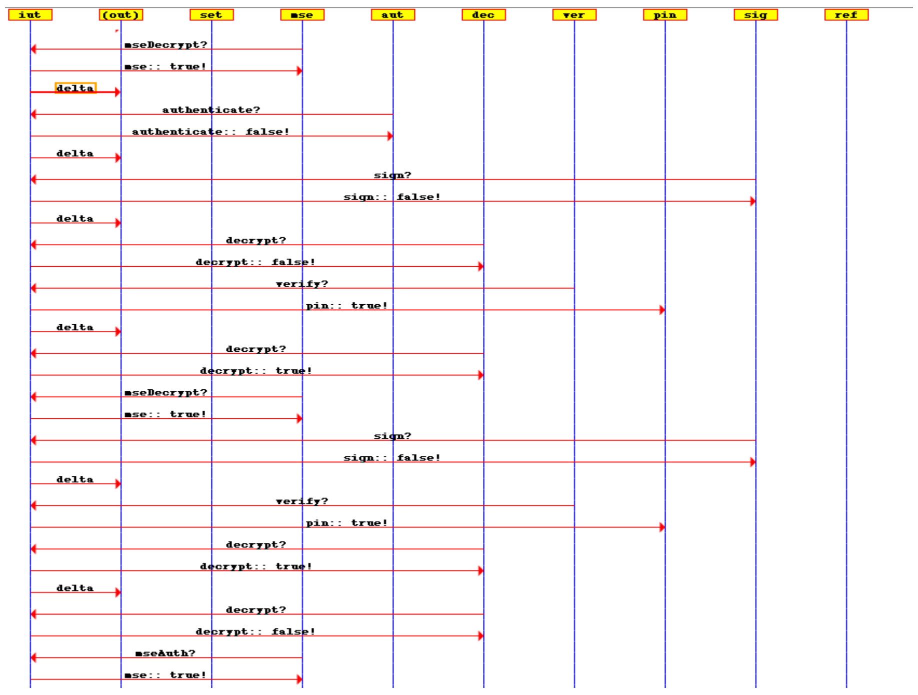

Figure 4: An extract of the message sequence diagram.

The finite state model that is generated shows all the paths JTorX took. As can be seen in figure 5, it allows us to see the choices in a more intuitive way than the sequence chart. Again, timeouts cause delta steps to appear in this model, these are also represented by circular arrows pointing to the state where the timeout occurred.

The finite state model has three main branches, and two small ones, just like Model 1 and Model 2. We can clearly see the otherwise nodes. Unfortunately not all the labels are readable, but this is due to way JTorX generates these models and this cannot be altered. We can see that every otherwise node has the edges described in the extension [figure 2]. Model 2 and this finite state model have an equal amount of nodes, an equal amount of edges and all the edges are connected to the same nodes. So they are equal.

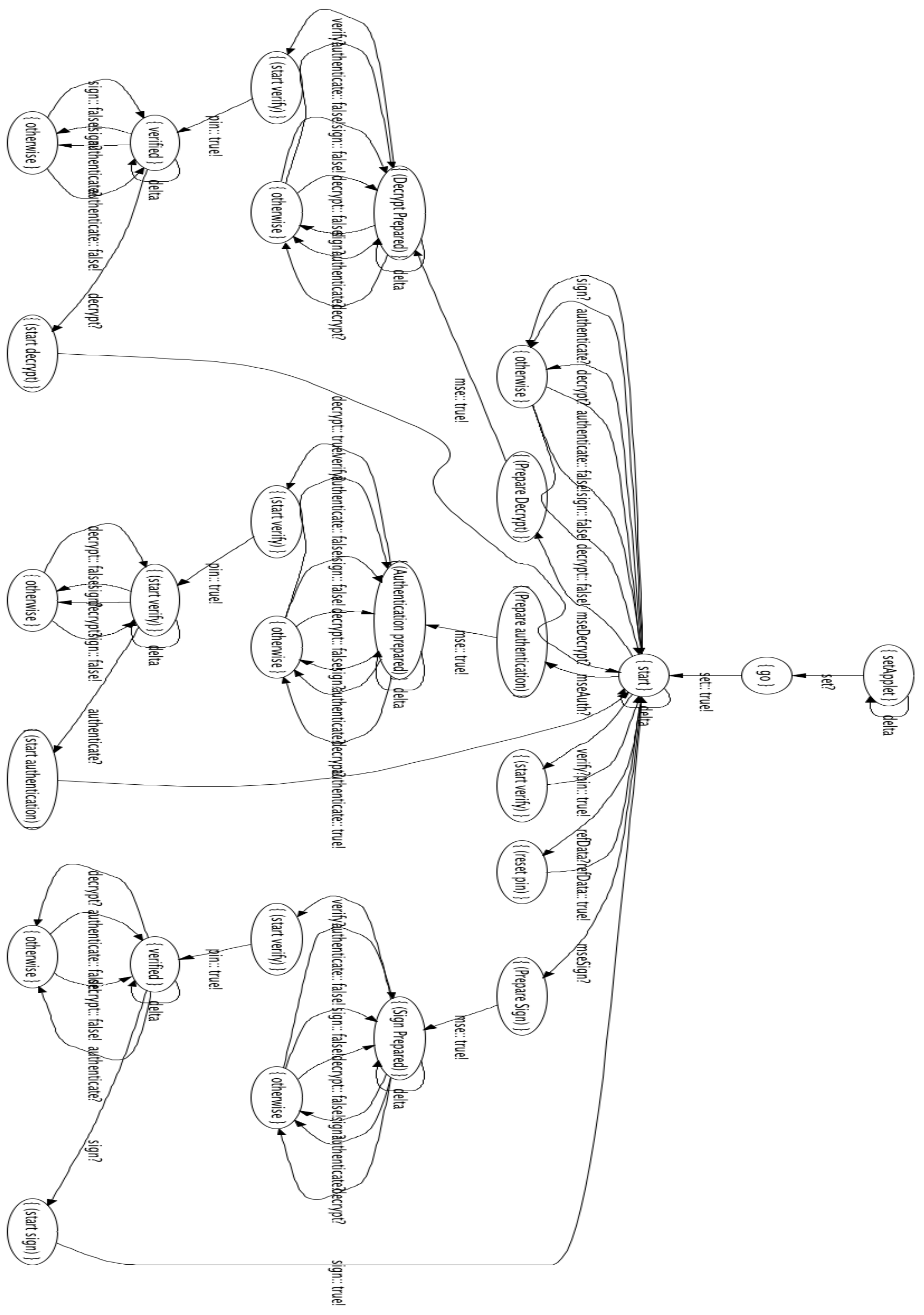

Figure 5: The generated finite state model of the test run.

7 Encountered problems

- The lack of proper documentation for Torxakis or JTorX made everything much harder. Good documentation of the tools functions combined with a tutorial would make it much easier to find out how things work. When getting started, there is so much that could cause an error (the model could be wrong, the adapter could be wrong or a setting in the tool could be wrong, etc) that it is hard to find what exactly is causing the error.
- Most Tutorials or examples have no description or documentation. However, there were a few tutorials for JTorX that had an adequate readme.
- JTorX has many options, that can be left at their default values. But all these option complicate things for someone that is new to model based testing.
- Model 2 required a lot of detailed work, even in a relatively small model like this it is easy to make mistakes, resulting in failed tests. Because the tests require a lot of time, this can be very time consuming. Attempting to edit the XML yEd creates was even more tedious. JTorX supports different input methods, and building the XML from scratch is perhaps the best solution for more complex models. It will make the modelling process less intuitive, but JTorX provides an interpretation of the model that is viewable as an image that should be sufficient.
- JTorX has a timeout but, unfortunately, it does not use this timeout, before attempting the next step. Only when the current step receives no response. Changing this could remove all delta steps. It would also significantly increase the time for testing, as all steps will at least require the same time as the slowest step, but perhaps this choice should be up to the user.
- JTorX does not record paths it has taken. It will always choose randomly. This causes JTorX to test paths it has already tested, or skip paths. To claim, with high certainty, that all paths have been tested at least once, thousands of iterations have to be taken. Many of these iterations are pointless.
- Uppaal was not meant as a drawing program. It has useful features to draw graphs, but other features, like automatic syntax checking, make it poor choice for this task. The reason for going with Uppaal initially was the export functionality, which could save time later. But this was ultimately never used, as instead the high level model was build from scratch.
- Creating the models shown in figure [8],[9] and [10] have cost a lot of time to make, and were never directly used in the testing process.

- For this thesis I considered two model based testing tools, JTorX and TorXakis. Both are based on TorX, which is developed by Tretmans et al. [8]. I decided to use JTorX because it was easier to setup than TorXakis. JTorX works out of the box, and the graphical user interface makes it easy to change settings. There is also a tutorial to guide new users through their first steps. TorXakis requires setting up, and only has a command line interface. Both tools lack proper documentation, and this makes it much harder to use a command line interface. TorXakis also seemed to have issues running on 64-bit Mac OS X. A key feature JTorX lacks when compared to TorXakis is random input during testing. However, as shown in section 5.3, this is not necessary as the adapter handles all input and output on an APDU level.

- Model 2 [figure 3] was particularly troubling. There was an error in the finite state model but I was unable to find the cause. The finite state model did not correspond with Model 2. The otherwise extension was causing odd behaviour. As can be seen in figure 6, a lot of extra nodes were created. Also, there are verify edges leaving otherwise nodes, and these edges are not present in Model 2. The sequence chart at an otherwise node did not show any odd behaviour.

  To fix this, the old extension [figure 7] was changed to the current one [figure 2]. Which is actually an equal and more elegant solution. The edges leaving the old otherwise node to the error state were made to direct the model to a state it could not leave, and cause a failed test. However, the test would already fail if the card would respond with anything other than a *sign ::false!, decrypt:: false!* or *authenticate:: false!*. This includes the *sign:: true!, decrypt:: true!* or *authenticate:: true!*, and thus the extra edges were superfluous. By removing these edges JTorX was able to generate a correct finite state model. I was not able to find out what was the exact cause of this problem.

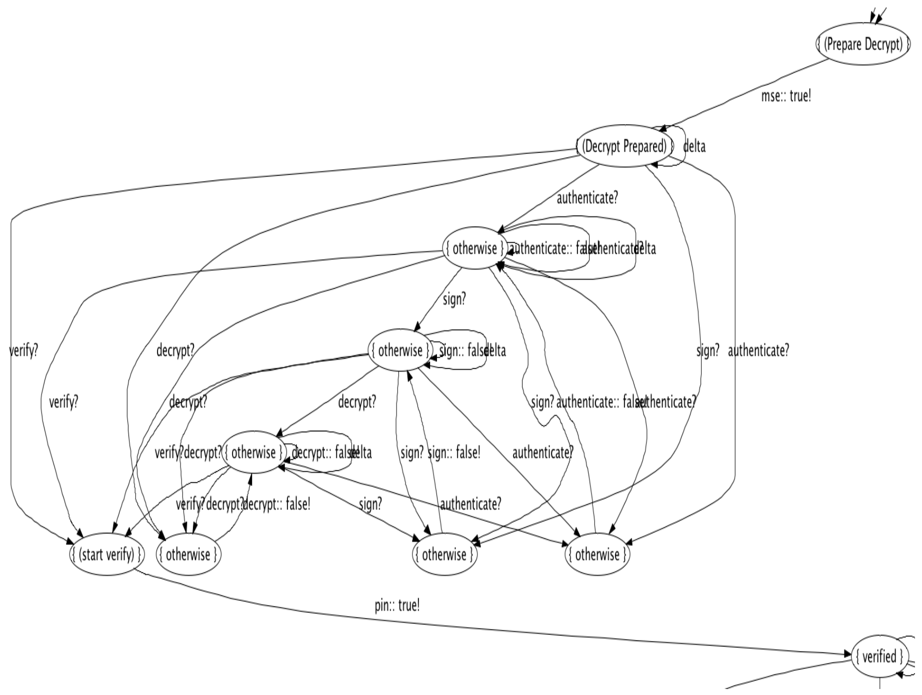

Figure 6: A small part of the finite state model generated by JTorX with an incorrect extension.

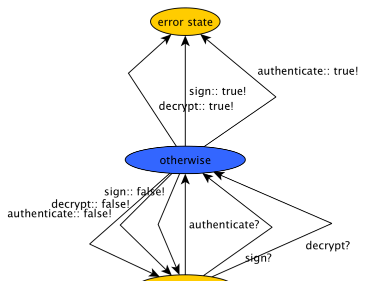

Figure 7: The incorrect extension.

8 Future Work

A possible option for future work is to test the host implementation for correctness and security. As this thesis only focussed on the on-card implementation. Another aspect is that not all commands were tested. All commands that are only used before personalizing were not tested. For example, it would be interesting to see if it is possible to personalise the card again, without losing the current keys.

9 Conclusion

Assuming Model 1 and Model 2 are correct, that JTorX and the adapter function correctly and that 10,000 steps is enough to try all possible options, then the javacardsign applet is correct and secure with respect to my models. As shown in section 6, the generated finite state model [figure 5] is equal to Model 2 [figure 3]. As can be seen in section 7 there was an issue with the extension I created, and how JTorX interpreted it. But this was the only case where JTorX interpreted the model differently than I expected.

The assumption that 10,000 steps is enough to try all possible options is difficult to condone. It would be much better if JTorX could simply test all options, instead of only supporting random testing.

Manually adding incorrect behaviour to Model 1, resulting in Model 2, was problematic. It has many details, and it requires analysing where the problems can occur, resulting in possible human errors. It would have been easier to let JTorX try some or all commands in every state, that were not specifically modelled. My otherwise extension was a crude way to achieve this, and ultimately caused a lot of time to get right. Although not all behaviour is tested, the important behaviour is.

A Appendix

For a better understanding of the code of the Javacardsign applet, I modelled some of its functions in greater detail than was necessary for the model based testing. I placed the analysis of three of these functions in this appendix.

A.1 Modelling the Protocol

When a card is produced and the software is installed, it is in the Initial state. It contains no personal information, and everything can be read. In this state, files can be written to the card, and the PUC and PIN can be set. Once this is done the card is personalised. When a card is personalised, the PUC and key files cannot be changed. Also, it is not possible to access certain files without a correct PIN.

The following models [8,9,10] only consider the parts of the code that is used if a card is personalised. The initial state of the card lacks any security measurements. The model in figure 8 is relatively simple, the following two become increasingly complex.

A.2 Verify the PIN

One of the more basic commands is the PIN verification shown in figure 8. Although important, the algorithm is rather straight forward. There are some possible exceptions being thrown, but no real branches exist. Likewise, the model represents the same property. This command's purpose is to compare stored reference data with verification data passed as a parameter in the APDU. In this case the stored PIN with the given PIN. The card keeps a record of unsuccessful attempts. When a certain limit is reached, the card will be blocked and requires the PUC code to be unblocked. All (default) values are in accordance with the ISO 7816 standard.[1]

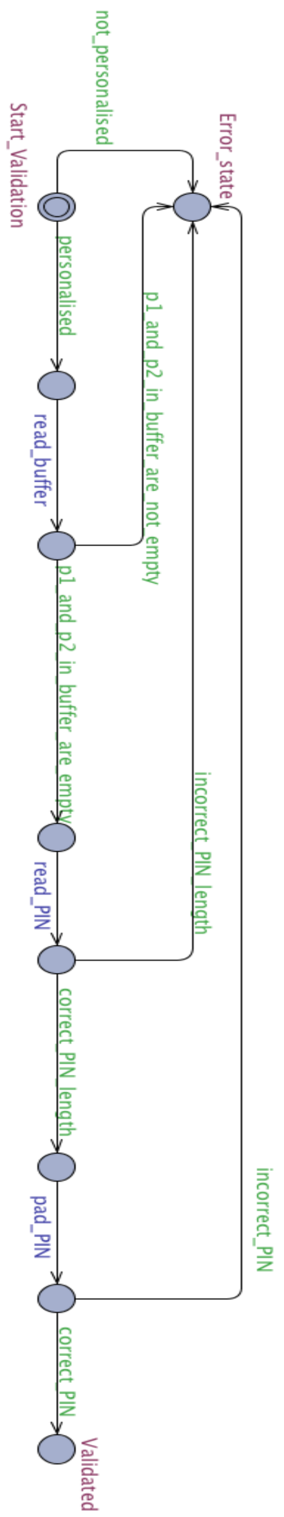

Figure 8: Verify instruction

The first check is to see if the card is personalised, because only in this state PIN checks are performed.

Then the buffer is read. The two bytes that are read should be 0x00. The byte P1 is unused at this time, but should still be 0x00. P2 shows which reference data on the card should be used. As the only available reference data is the PIN, this should always be 0x00.

After that the LC data is read, and this contains the verification PIN. The first check on the verification PIN is a length-check. If the length is too short or too long, the card will end the VERIFY instruction, with no consequences for the amount of attempts that remain. If the length is correct, the data is prepared. The card then compares the verification data with the reference data. On a correct PIN, the result is a successful end of the algorithm. On an incorrect PIN, the algorithm ends unsuccessfully and one less verification attempt is available.

A.3 Change Reference Data

The Change Reference Data instruction can do two things. It replaces the reference data stored on the card. If the card is in the production state, this command is used to set the PUC. If the card is in the personalised state it can be used to change the PIN if the PUC is provided. In this project we only look at the personalised state, so only the second option path is represented in figure 9.

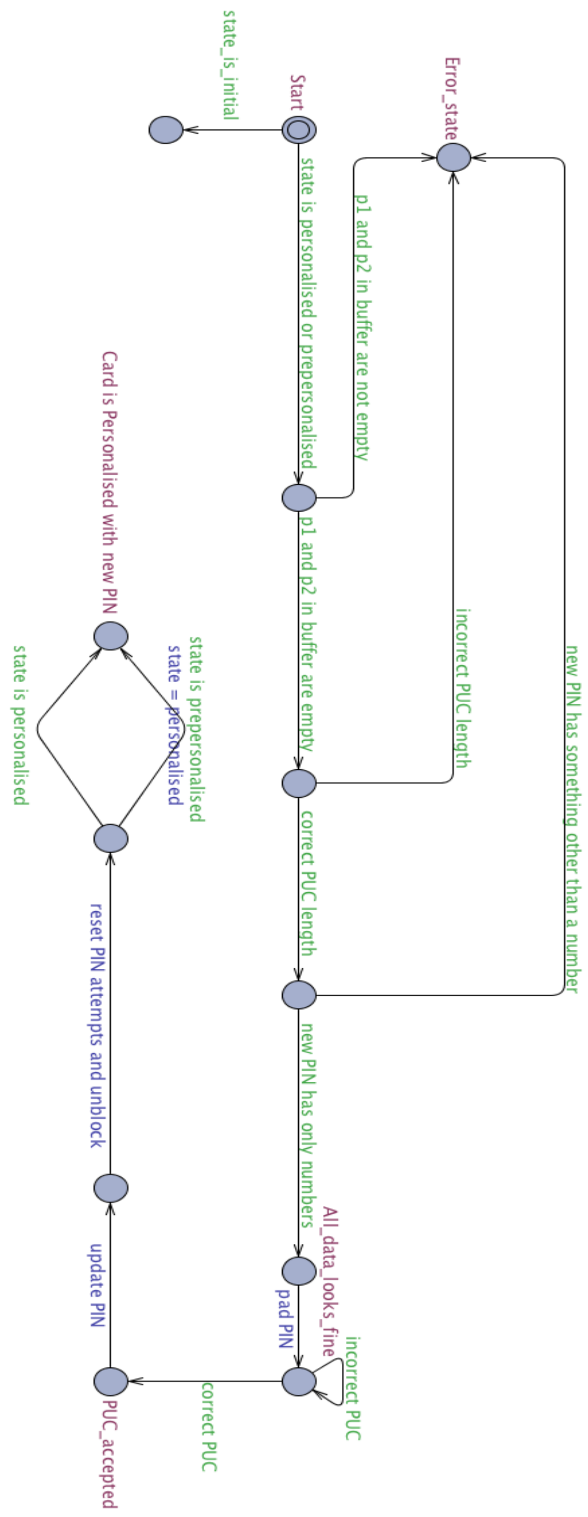

Figure 9: Change Reference Data instruction

It starts with reading the buffer. Assuming we are in the personalised state, the bytes P1 and P2 are unused, but should be 0x00 nonetheless. Other values are reserved for future use, and could cause problems with future devices. Because this is in the personalised state, the only thing that can be changed is the PIN. To be able to change the PIN the PUC is required. The first check on the PUC is the length. There is a minimal and maximum length for the PUC. An entry with the wrong length aborts the instruction.

The second check is to make sure the PUC only contains numbers. Although the bytes are checked one by one, this is represented by one step in the model. A byte anything other than a number will abort the instruction. Once the it is verified that the length is correct, and the PUC only contains numbers, the PUC will be verified. A correct PUC will continue the instruction, while a incorrect one will abort it. Once the PUC has been verified, the reference PIN is updated with the new PIN. The card is then unblocked if it was blocked and the amount of attempts to enter the wrong PIN is reset. If the card was not already in the personalised state, it is set in that state. If it was already in that state, nothing changes. This will end the instruction.

A.4 Select File

The select instruction prepares a file on the card to be read. In this model, and in the code, EF stands for Elementary File, DF stands for Directory File and MF stands for Master File. Using these parameters, it can prepare different types of data files with different methods. The file location should be in the LC data, and corresponding with the Master File IDs. The particular method (direct, under current DF, parent of current DF) of selecting a file is based on the value of P1. P2 can be arbitrary.[1]

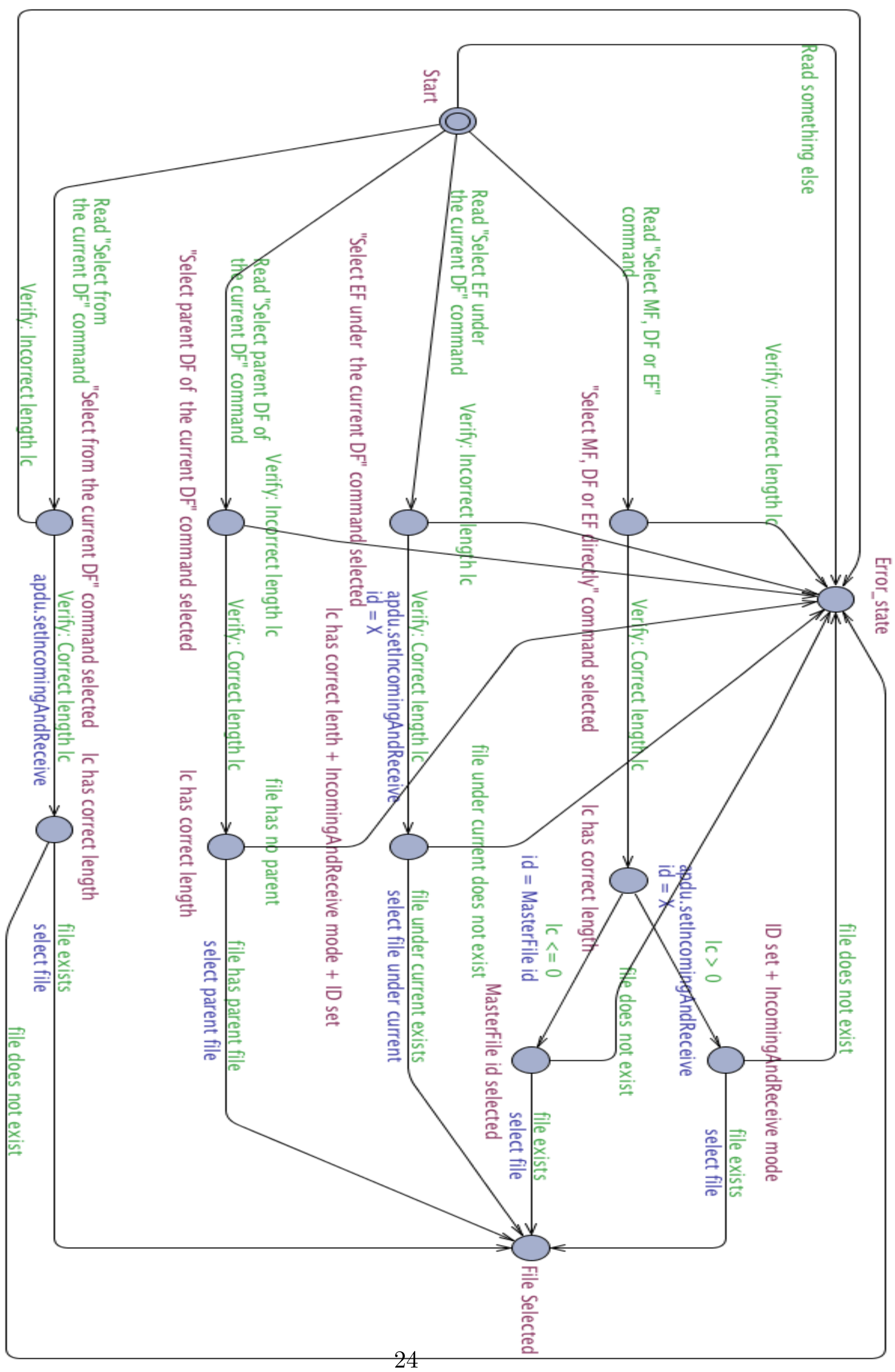
Start
Read something else
Read "Select MF, DF or EF" command
Read "Select EF under the current DF" command
Read "Select parent DF of the current DF" command
Read "Select from the current DF" command
"Select MF, DF or EF directly" command selected
"Select EF under the current DF" command selected
"Select parent DF of the current DF" command selected
"Select from the current DF" command selected
Verify: Incorrect length lc
Verify: Incorrect length lc
Verify: Incorrect length lc
Verify: Incorrect length lc
Error_state
Verify: Correct length lc
Verify: Correct length lc
Verify: Correct length lc
Verify: Correct length lc
apdu.setIncomingAndReceive
id = X
apdu.setIncomingAndReceive
id = X
apdu.setIncomingAndReceive
lc has correct lenth + IncomingAndReceive mode + ID set
lc has correct length
lc has correct length
lc has correct length
file under current does not exist
file under current exists
select file under current
file has no parent
file has parent file
select parent file
file exists
select file
file does not exist
lc <= 0
id = MasterFile id
lc > 0
MasterFile id selected
ID set + IncomingAndReceive mode
file does not exist
file exists
select file
file exists
select file
file does not exist
File Selected

Figure 10: Select File instruction

The select instruction appears a lot bigger than the others, with this much detail. However some recurring elements can be seen in the model. The instruction starts by accepting one of the four different possible commands, as can be seen from the four different paths from the starting position. If none of these four commands is recognised, the algorithm aborts. This is represented by the loop at the starting position. Once a command is selected, all branches generally continue in the same way.

If the LC data has a correct length, it is interpreted as a file location. If the length is incorrect, the algorithm aborts. The file location in the LC data is read, and if it is a valid location the file is prepared. If the file cannot be found an exception is thrown and the algorithm is aborted.

An exception for this is the branch for the "select MF, DF or EF"-command. This will select the Master File by default if no parameter is given. There are no checks on the input data of P2. This selection can be seen by the split at the end of the branch.

References

[1] ISO/IEC 7816 - Smart Card Standard, 2000.

[2] Nederlandse Vereniging voor Burgerzaken (Dutch Association for Civil Affairs). e-NIK Functionality `http://www.nvvb.nl/websites/nvvb/website/default.asp`, March 2011.

[3] W Mostowski. Java Card PKI `http://javacardsign.sourceforge.net/`, March 2011.

[4] E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the JavaCard API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.

[5] W. Mostowski, E. Poll, J. Schmaltz, J. Tretmans, and R. Wichers Schreur. Model-based testing of electronic passports. *Formal Methods for Industrial Critical Systems*, pages 207–209, 2009.

[6] W. Mostowski and E. Poll. Electronic passports in a nutshell. Technical report, Technical Report ICIS–R10004, Radboud University Nijmegen, June 2010. Available at `https://pms.cs.ru.nl/iris-diglib/src/getContent.php`, 2010.

[7] A. Belinfante. Jtorx: A tool for on-line model-driven test derivation and execution. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–270, 2010.

[8] Alex Belinfante, Ed Brinksma, Jan Feenstra, Jan Tretmans, and René Vries. Côte de resyste : Automatic model-based testing of communication protocols. In *7th Annual CTIT Workshop on Mobile Communications in Perspective*, pages 49–51, 2001.