Master Thesis

Java card software analysis using model checking

Author:

Hugo Brakman

Supervisors:

dr. ir. Erik Poll
dr. Wojciech Mostowski
prof. dr. Frits Vaandrager

April 5th 2007

Research number 568

Radboud University Nijmegen

Abstract

Smart cards are used for a number of purposes. They are fitted on bank cards, used in mobile phones and many other products. The cards often contain confidential or secret data. A high level of security is therefore required. However, smart cards are easily obtained by assailants and tampering with them can be as easy as removing the card from its external power supply half way computation. The implementation of the software operating on the card can sometimes provide security against attacks on the hardware.

We consider Java cards, smart cards that allow for the execution of Java applications. In this thesis we investigate the use of model checking in combination with fault injection in verifying the robustness of software on the Java card against different attacks on its hardware. We implemented an experimental tool that automatically generates a model from Java code, allowing for attacks to occur. We used this tool to generate models to verify the security of our running example, a PIN implementation, against different attacks. This supports our idea that automatic generation of such models directly from code can help developers to speed up the process of creating robust implementations that can withstand hardware attacks.

Contents

1 Introduction **9**
- 1.1 Cards . . . 9
- 1.2 Modelchecking . . . 11
  - 1.2.1 What is modelchecking? . . . 11
- 1.3 Motivation, goals and approach . . . 12
- 1.4 Structure of this document . . . 13
- 1.5 Related work . . . 14
- 1.6 Acknowledgments . . . 14

2 Background to the modeling **15**
- 2.1 A brief introduction to the smart card . . . 15
  - 2.1.1 Components of the smart card . . . 16
- 2.2 Attacks on smart cards . . . 17
  - 2.2.1 Logical attacks . . . 17
  - 2.2.2 Physical attacks . . . 18
  - 2.2.3 Side channel attacks . . . 19
  - 2.2.4 Attacks considered in this research . . . 20
- 2.3 Running example . . . 21
- 2.4 Jabstract . . . 24
- 2.5 SMV . . . 24
  - 2.5.1 Why SMV . . . 24
  - 2.5.2 Discrete time . . . 25
  - 2.5.3 The SMV language . . . 26
  - 2.5.4 Properties . . . 28

3 Fault injection models from Java code **33**
- 3.1 Overview . . . 33
- 3.2 Modeling Java in SMV . . . 34
  - 3.2.1 Data types . . . 35

3.2.2 Arrays . . . 36
3.2.3 Names and hierarchy . . . 37
3.2.4 Sequential code . . . 40
3.2.5 Method structure . . . 41
3.2.6 Expressions . . . 49
3.2.7 Blocks of SMV and jumping . . . 50
3.2.8 Examples of statements . . . 52
3.3 Modelling attacks . . . 57
3.3.1 In time attacks . . . 59
3.3.2 Framework . . . 66
3.3.3 Constructor . . . 67
3.4 The SMV Generator . . . 68
3.5 Verifying Properties . . . 72
3.5.1 Secure PIN . . . 73
3.6 Problems faced . . . 75
3.6.1 Syntax tree . . . 75
3.6.2 Lists . . . 76
3.6.3 Visibilities . . . 77
3.6.4 Expression result . . . 78
3.6.5 State space . . . 79
3.6.6 Small specific problems . . . 79
3.7 Used tools . . . 84
3.7.1 SMV . . . 84
3.7.2 Jabstract . . . 84
3.7.3 Java environments . . . 84
3.7.4 PC and software . . . 84

4 Intuitive and defensive PIN implementation 85
4.1 Java code . . . 85
4.2 Verifying the implementations . . . 87
4.2.1 The secure property . . . 88
4.2.2 Verification of secure . . . 88
4.2.3 Continue attacks on the defensive implementation . . . 92

5 Discussion and conclusion 97
5.1 Discussion . . . 97
5.1.1 Alternative approaches . . . 99
5.2 Conclusion . . . 100

6 Future work **103**
6.1 In general 103
6.2 Continuing this research 103
6.2.1 Objects 104
6.2.2 Dependencies within Java 104
6.2.3 Improvements on the generator 104

Appendices **109**

A Generation example **109**

Chapter 1

Introduction

1.1 Cards

Smart cards are computers so small they can be fitted on bank cards and many other products. It is likely there is one in your pocket right now. Because of their versatile usage, very small size and low costs, they are used more and more often. Smart cards are definitely gaining ground and will probably continue to do so. ([BBE+99], [LAMD04])

A smart card on a credit card sized piece of plastic looks like this:

Because the cards are also used for security related purposes, the importance of their security aspects is increasing. That is where a number of problems arise. The cards are literally in the hands of possible assailants (or could easily get into their hands). This means they can do just about anything with them physically and there is hardly anything the manufacturer can do to prevent them from trying. Also they are cheap to buy as blank cards and therefore the assailants can get their hands on other cards they can use as a reference to try different attacks without destroying their original cards.

An example of an attack could be as simple as a person running a card over by a car to see if that would break security. This is probably not the

most effective attack but the card will probably respond differently afterwards (if at all). Also more threatening attacks can be attempted ranging from putting the card in a microwave, moving it through a high magnetic field, shooting it with a laser even to scratching it open and tinkering with the internal components of the card.

Now it is very hard to protect a smart card properly. Consider the following simplified example of code where the triesCounter keeps track of the number of tries left for a person to try a personal identification number (PIN). The idea is that it will block the card after, say, three consecutive incorrect tries. Limiting the consecutive tries is a way of maximizing the risk of a correct guess to roughly $\frac{3}{10000}$. At least it is with a four-digit PIN and a maximum of three tries.

The following example shows the intuitive implementation of such a protection:

Intuitive implementation:

```
if(pincode not correct)
{
        decrease triesCounter
}
```

Note that here and throughout the rest of this writing we use mainly snippets of code instead of the full versions.

The example above might look fine but this implementation is not entirely safe. It allows for the following attack:

After the PIN is checked, the result of the check might be known outside the card (and therefore by an assailant). In section 2.2 we describe more on how this can be accomplished. When a PIN is checked there are two options. Either it is correct or it is incorrect. If it is correct, the assailant might know the PIN in which case he or she can do whatever she wants with it. Now suppose the PIN is incorrect. If, right after the PIN is checked but *before* the triesCounter is decreased, the card is removed from its powersupply, the counter will not be decreased. This means that the incorrect PIN attempt is not "recorded" and that the assailant can try the trick again. With this trick the person would have a theoretical infinite number of attempts to try the PIN therefore breaching security in a major way.

This is not an arbitrary example. It is hard to protect a card from a power cut in hardware but, in this case, perhaps that is not even required. If the software is written with more caution perhaps it would look like this:

Defensive implementation:

```
decrease triesCounter;
if(pincode is correct)
        set triesCounter to maximum;
```

In this example if a person cuts off the power, right after the check of the PIN, the counter has already been decreased. In fact in this way no information is leaked before the decrement of the counter. Cutting off power here is completely ineffective as an attack and, in fact, would even block the card if a *correct* PIN is entered sufficiently many times.

This example shows how in some cases, even though hardware based protection is hard or impossible, software protection can be very effective in securing matters.

Now to get to the topic of this research. It is not easy to see with human eyes where weak points in a piece of code exist. What is a weak point anyway? In order to assist the developer in writing code that can withstand certain attacks it is perhaps possible to use techniques such as model checking.

This research investigates the use of model checking to verify properties of code for smart cards. In particular it focuses on *Java cards*[1], smart cards that allow for the execution of Java code. However, the results could be used for other purposes such as other smart cards or other code verification demands.

1.2 Modelchecking

Implementing software that prevents the exploiting of hardware weaknesses can be very hard since it is not obvious, for humans, where the weaknesses in code exist. Furthermore most of us are not trained to be aware of such hardware vulnerabilities while implementing code. This is where model checking could provide some help. Through the use of model checking it is possible to search for weaknesses in implementations or, if the modelchecker itself is trustworthy, even proof the tamper-resistance of code.

1.2.1 What is modelchecking?

Model checking is the process of checking whether a given formal model satisfies a given formal formula. Consider the following small example:

[1] http://java.sun.com/products/javacard/

Simple model checking example:

```
begin:
a = 0;
allowed step:
a = a + 1
```

This is not an existing language, it serves merely as an example of a model, but it is supposed to state that initially a has the value 0 and that with each step a is increased. Therefore a will have the following values in order: $0, 1, 2, 3, 4, \ldots$.

In this model, where only a and its rules exist, one can state the property that "always at some step a will be 30". A model checker could than take the model as well as the formula describing that property and try to check whether or not the property holds for that model. If it does not, often a model checker can give a counter example.

You could also come up with more interesting things to check such as

- "always a is greater than -1" or
- "at all steps a will, at some next step, have the value of a at the current step to the power of a at the current step" or
- "at any step there will be some next step where a is prime".

The important thing about model checking is that a computer can do it for you once you have a correct model and property. There are of course limitations, but a modern model checker can verify complex properties on complex models.

For this research we focused on the SMV model checker, which will be discussed further in section section 2.5.

1.3 Motivation, goals and approach

The main goal of this research is to to investigate the use of model checking for formally verifying resistance of Java card applications to side channel attacks, in particular fault injections.

We consider model checkers mostly because of their automation and their possibility to reason about non-deterministic events.

Initial work has been carried out by Hubbers, Mostowski and Poll when they investigated card tears, a fault injection attack, in [HMP06]. They used the Uppaal ([BDL04]) model checker to verify resistance of their code

to these attacks. They encountered two main disadvantages of their use of Uppaal:

- The manual creation of the model results in a messy state chart diagram which is unmaintainable
- Only one kind of fault injection is considered

The first disadvantage is mainly because the model of the Java code will be large and hard to design and modify using the graphical interface.

With this initial single case study in mind our work contributes (at least) in the following ways:

- An automatic translation of Java into the SMV model checking language
- A (more or less) universal way of modeling different kind of faults
- The use of a more appropriate model checker

Doing this required

- a careful study of how to translate Java into the SMV language (this is not a straightforward process)
- to automatically generate the SMV language and keep the generated model manageable (to make verification possible)
- to decorate the generated SMV with a model of the attacks that we want to investigate
- to consider how to allow for the specification of properties we want to verify
- assumptions on certain properties of the translated code and certain limitations of the whole approach.

1.4 Structure of this document

After this section the document will first describe some background information in chapter 2, including the example on which we focused during this research. The next chapter then continues with the story of SMV generation from Java code and ends in details about the experimental SMV generator we wrote as a proof of concept.

In chapter 5 we discuss the research and the implications. We used the generator on the running example which is described in chapter 4.

After that we state the conclusion of this research in section 5.2 and end with suggestions for future work in chapter 6.

In the appendix a small example of Java code and the actual generated model is provided.

1.5 Related work

In [HMP06] Hubbers, Mostowski and Poll report on investigations into the Java card transaction mechanism. As part of their work they created a model to investigate the robustness of code against card tears. We used their work as a starting point (section 1.3) and running example (section 2.3).

In the SLAM project of Microsoft[2] researchers aim at automatic checking of whether or not programs obey "API usage rules" (Rules that specify what it means to be a good client of an API). Specifically they investigated Windows device drivers. For a brief introduction we refer to [TB02].

A different approach from formal verification is simulation. In [RNS+04] a methodology for high level fault injection for security attack simulation in smart cards is presented. The authors suggest to consider a smart card representation as different functional blocks. Different kinds of memory and other components of the smart card form these blocks. In between these blocks faults can be inserted. The authors start by considering a design of a system and then create a second design from it by injecting faults into it. Then they simulate the behaviour of both designs and compare and analyse the results.

1.6 Acknowledgments

First of all I would like to thank Erik Poll and Wojciech Mostowski of the *security of systems* group and Frits Vaandrager of the *informatics for technical applications* group for their supervision and support.

There are many people that have helped me in many different ways, ranging from discussing the research and providing moral support to distracting me and forcing me to take a break every once in a while (the cats at home show no mercy). I would like to thank everyone that helped me in any way. Thank you Eva.

[2] http://research.microsoft.com/slam/

Chapter 2

Background to the modeling

In this chapter we will give some more background information on

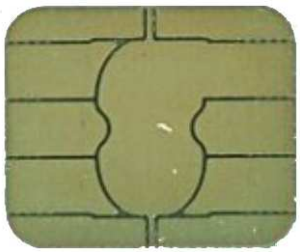
1. smart cards (section 2.1),
2. possible attacks (section 2.2),
3. the example we focused on (section 2.3),
4. Jabstract (section 2.4), the parser we used as a base for the SMV generator,
5. the SMV model checker (section 2.5),

In the next chapter this information is used to describe the process of generating an SMV model from Java code.

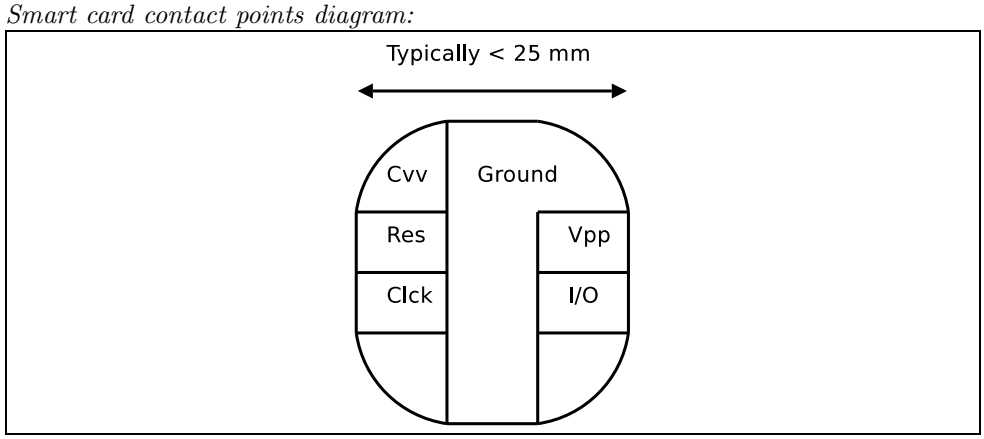
2.1 A brief introduction to the smart card

The smart cards, as most of us know them, have a layout illustrated in the next images.

Smart card contact points:



Smart card contact points diagram:



In the last image

- Vcc and Ground are used to provide power
- Res allows a hard restart of the processes
- Clck is where the clock signal is provided
- Vpp is no longer in use
- I/O is used for the communication

For me realizing that these contact points are waiting in my wallet for input made me also realize that indeed anyone can try an attack on these computers.

The image of the smart card above is not the only existing smart card type. There are also contact-less variants of the smart card used in, for example, badges and other items.

2.1.1 Components of the smart card

The basic components of the chip are the

- Central Processing Unit (CPU)
- Test Logic unit that is used only during production to test the chip
- Security Logic unit that checks the environmental conditions
- I/O interface that allows for serial communication

- ROM memory that contains all the permanent memory such as the operating system
- RAM memory that is used to store all kinds of temporary data
- EEPROM updatable memory that even retains the stored information when there is no power, which contains keys such as the PIN and sometimes application code and other data

These components are connected via the data bus.

Looking at the components, you see the similarities and differences with regular computers.

2.2 Attacks on smart cards

In this section we will briefly describe some of the common threats to smart card security. A more elaborate discussion on these threats and more can be found in [Wit02].

The attacks on smart cards can be divided into different categories. In the next section we briefly go over some of the attacks from each of these categories. In this research we do not go into all the details of these attacks but it is important to have some understanding of the possibilities and to realize that these attacks are not just theoretical.

2.2.1 Logical attacks

Logical attacks are based on bugs in the implementation of the software on the smart card. Hidden commands might still be active from a previous phase such as the initialization phase. This can allow someone to execute them after initialization, when they are not supposed to be executed. Also communication protocols and their implementations might contain bugs that can be exploited. It might be possible to trick the card in sending secret information for example. Obviously the cryptographic system is vulnerable if it contains bugs.

These are just some examples of logical attacks. One of the important aspects of these attacks is that they are not exclusive to smart cards but in fact are common attacks on about any computer. In this research we will not look at typical logical attacks since it is not about bugs but weaknesses. The difference we are looking at is that a bug can be exploited without the explicit use of certain properties of the hardware. When only software is

concerned, code might be free of bugs but can still be vulnerable to attacks on the hardware.

2.2.2 Physical attacks

The category of physical attacks contain the attacks that use the analysis or modification of hardware. It is often possible to use certain physical techniques to gain information on the design or even data of the chip card. The most obvious example is scratching the surface open to take a look "inside" the chip. This is in fact a serious threat, depending on what can be seen "inside". It might be possible to discover the actual layout of the chip by looking at it under a microscope. If you can find the wires of the bus for example, it is interesting to know what data travels on it. It is not hard to imagine that security sensitive data travels from memory to some other component. To do this, it is convenient to have probe needles placed on the wires to simply read the data.

The chip industry knows about this threat and develops chips with smaller and smaller components using many layers of hardware with components that do not longer have a clear layout but are mixed with the other components to hide their layout.

On the other side techniques are developed to deal with the smaller components and today even techniques such as the focused ion beam can be used. This is a variation on the scanning electron microscope which shoots ions making it possible even to deposit material on the tiny surfaces of the wires to rewire the chips circuit.

Another example of a physical attack is the use of certain chemical solvents, etching and staining materials. These can sometimes be used to de-layer the chip with high precision. This can help the methods in the previous paragraphs to get to the interesting components. Staining can be used to even make the difference in material that distinguishes the ones from the zero's visible. You can then read the actual memory.

One of the important side effects of these physical attacks is that the smart card itself is damaged.

Both the advanced attacks and the countermeasures require some sum of money but, considering the gain in breaking security, there will be people willing to invest money on such attacks. However, the different smart card manufacturers might not all spend money to install all the latest expensive countermeasures.

A less expensive, and completely different countermeasure that is typically used, is to keep the ownership of the card to the distributor and having

the smart card user to accept some legal conditions before even providing the card. This discourages arbitrary physical tampering but, of course, will not stop any serious criminal.

2.2.3 Side channel attacks

The possibilities of side channel attacks should not be underestimated. These attacks do not *physically* modify or analyze the card but use physical phenomena to analyse or modify the smart card *behavior*.

Typical properties that can be used for *analysis* (most of them are also used against other computer types) are:

- Power consumption. The components of the chips require power to operate. Depending on which component is doing what, the required amount of power differs. This can give an indication on the outside of the chip of what is going on inside the chip. A strong method that uses this is *differential power analysis*.

- Electromagnetic radiation will be transmitted by the smart card also depending on the internal processes. This again can be measured and possibly leak information to the outside of the chip.

- Time is yet another threat to security since typically it depends on what is going on inside the chip what amount of time it takes for the chip to do something measurable. Even something as simple as a reply on the I/O channel can in this way leak information.

The smart card might be sophisticated, but it still is a small computer that requires power to operate and it is not resistant to all environments typically threatening to any computer.

- The chips are sensitive to the voltage they operate on. Providing too high or too low a voltage might trick the chip in behaving differently. Power glitching is an effective technique where the chip will undergo a sudden change in power. This might cause the device to shut down[1] or even allow the card to continue after an incorrect execution of some operation.

[1] If suddenly the power is cut off (card-tear) most cards will stop halfway execution, causing abnormal behavior in some cases such as incorrect values being written to memory [HMP06].

- Most computers are sensitive to electromagnetic radiation because it can induce signals into the wires which can result into changing the behavior.
- Light and X-Rays can cause different behavior inside the chip. It sometimes is possible to "edit" the memory of the chip in this way.
- The chips are designed to operate under some temperature range. When the chip is used outside this range possibly the behavior changes.
- As depicted in the image on page 16, the clock signal comes from outside the chip. This means it is easy to change it at any time to any frequency. This used to be to great advantage of assailants because they could monitor the chips operations, one at the time, by setting the frequency low enough.

Obviously also against these attacks hardware measures are being developed. The most obvious is perhaps the Security Logic unit (section 2.1.1) in which different sensors monitor the environment for changes in frequency, temperature and probably many other threats.

Especially differential power analysis and power glitching is used a lot to change the behavior, probably due to the low costs and the lack of knowledge required to use it. Power glitching can be as easy as removing your card from a vending machine half way calculations. Of course knowledge will help in attacking these cards, but it does not compare to, for example, the expertise required to reverse engineer the chip.

2.2.4 Attacks considered in this research

There are many different kinds of attacks, some of which require different countermeasures. The focus in this research is on those attacks that trick the card into incorrect behavior. Specifically those that make a read or write operation fail. Some examples therefore might include the use of light to modify memory (causing read operations to result in a value that has not been written according to the code) and power glitching that might cause incorrect write operations.

Also, in the example we focus on (section 2.3), we assume the possibility of obtaining information from the card, such as the correctness of the PIN during or after the required calculations, to make that decision. To do this, possibly time can be used to analyse the calculation or perhaps some other method can provide such information.

2.3 Running example

We already explained the basic ideas of the example on page 10.

The code that checks the PIN can be as follows:

Intuitive implementation:

```
if(pincode not correct)
{
        decrease();
}
```

or preferably like this, which is more secure:

Defensive implementation:

```
decrease();
if(pincode is correct)
        max();
```

This code is only supposed to be executed if the card is not blocked. This can be checked using the counter that registers the number of tries left.

The example comes from [HMP06], which in its turn served as a starting point for this research. We used the original Java code and model that Hubbers, Mostowski and Poll used. The model was created with the model checking tool Uppaal[2] [BDL04].

Due to the transaction mechanism on the Java card it is not safe to store the counter in a regular variable. The counter is instead stored in an array, so it can be updated using the non-atomic method arrayCopyNonAtomic, to bypass the transaction mechanism. The security threat to using this non-atomic method is that an interruption of it can cause the counter register to get a random, possibly very high, value. In order to prevent this from breaching security it is possible to use three different arrays, instead of one, and the methods below to store and update the counter.

The following Java snippets contain the most crucial parts of this code which is used by the PIN verification code above. The complete version of the Java code is listed in chapter 4. In this code the variables $\_c1$, $\_c2$ and $\_c3$ represent the different variables of the counter, used to indicate the number of PIN attempts left. These methods are required for the PIN verification examples as shown above. We will now show the Java code followed by a brief description of it.

[2] http://www.uppaal.com

TryCounter snippet: setNonAtomic:

```
private void setNA(byte value) {
        _temp[0] = value;
        Util.arrayCopyNonAtomic(_temp, (short)0, _c1,
                (short)0, (short)1);
        Util.arrayCopyNonAtomic(_temp, (short)0, _c2,
                (short)0, (short)1);
        Util.arrayCopyNonAtomic(_temp, (short)0, _c3,
                (short)0, (short)1);
}
```

The *setNA* method is used to store the counter value in all of the three counter variables. It is important to write to the variables in this order because the restore method depends on this.

The array copying method comes from the javacard package and is used to copy an array, bypassing the transaction mechanism.

TryCounter snippet: get:

```
public byte get() {
        restore();
        if (_c1[0] == _c2[0] && _c2[0] == _c3[0])
                return _c1[0];
        return (byte)0;
}
```

The *get* method is used to retrieve the correct counter value from the three backup variables.

TryCounter snippet: decrease:

```
public void decrease() {
        restore();
        if(_c1[0] == (byte)0) {
                return;
        }
        setNA((byte)(_c1[0]-1));
}
```

The *decrease* method is used to decrease the counter value (which should happen when the incorrect PIN is attempted).

TryCounter snippet: max:

```
public void max() {
        restore();
        setNA(_max);
}
```

The *max* method is used to reset the counter value in case the PIN is correct.

TryCounter snippet: restore:

```
private void restore() {
        if(_c1[0] == _c2[0] && _c2[0] == _c3[0]) {
                return;
        }
        if(_c2[0] == _c3[0]) {
                _temp[0] = _c3[0];
                Util.arrayCopyNonAtomic(_temp, (short)0,
                        _c1, (short)0, (short)1);
                return;
        }
        if(_c1[0] == _c2[0]) {
                _temp[0] = _c1[0];
                Util.arrayCopyNonAtomic(_temp, (short)0,
                        _c3, (short)0, (short)1);
                return;
        }
        // All three different
        _temp[0] = _c3[0];
        Util.arrayCopyNonAtomic(_temp, (short)0, _c2,
                (short)0, (short)1);
        Util.arrayCopyNonAtomic(_temp, (short)0, _c1,
                (short)0, (short)1);
}
```

This *restore* method is used to restore the correct counter value from the backup variables. The basic idea is that you do not write to the backup variables that are assumed to be correct before the writing to the (assumed) incorrect backup variables has succeeded.

This, in total, is the code we took as a point of reference.

2.4 Jabstract

Jabstract: A full Abstract Syntax and Parser for Java[3] is a project by Sebastian Danicic. The current version (section 3.7) is written for Java 1.1 which is sufficient for the kind of code this research is about since Java cards do not, at the moment, allow sophisticated Java features.

The system was implemented using the JavaCC^{TM4} (Java Compiler Compiler) tool.

The project is not claimed to be fully accurate but when we manually reviewed and tested the abstract syntax tree it generated for different Java testing code it seemed to work at least sufficiently well for writing a "proof of concept" SMV generator.

The project has no documentation other than in the form of Javadoc and consists, apart from the Javadoc, only of Java classes.

2.5 SMV

The problem with model checking is that usually there are too many possibilities to simply go over them. Somehow the possibilities (states) have to be represented and handled in a smart way to handle their huge number (state explosion). The problem of how to do that is known as the state explosion or combinatorial explosion problem.

Most methods, such as temporal logic model checking, language containment algorithms for automata and others, explicitly represent a state space by means of a list or table which grows in proportion to the number of states ([JRBH92]). SMV (Symbolic Model Verifier), as some other model checkers, uses a different technique called symbolic model checking. This technique uses boolean formulas by which it represents sets and relations.

In this section we will mainly go over parts of the SMV language to support the examples in the upcoming sections.

2.5.1 Why SMV

SMV is used a lot with success and, because of its reputation, in combination with the experience of Frits Vaandrager with it, we decided to test it first for this research. After it "passed" the initial tests to see if it was suitable we decided to use it for this research. If it would not have worked out at

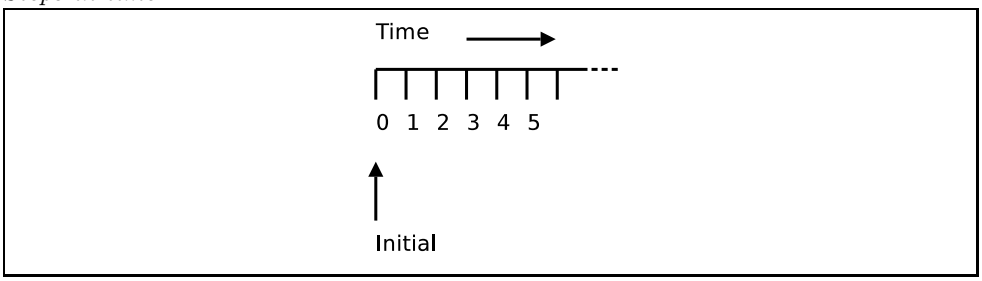
[3] https://jabstract.dev.java.net/
[4] https://javacc.dev.java.net/

the end, a lot of ideas, as will be explained in the rest of this chapter, could still be translated into other model checking languages.

2.5.2 Discrete time

The language used to describe a model for SMV allows definitions of transitions of variables over discrete steps in time.

Steps in time:



The ways to do this are:

init	init(a):=0	defining the initial value of some variable (at time 0 in the figure above)
next	next(a):=a+1	defining a to have the current value of a plus 1 as its value in the next point in the discrete timeline.
	a:=0	defining a to be 0 at the current time

A note on the next function. When considering *next(a):=0*, it is assumed that a will *continue* to have the assigned value, even after the next point in time (unless explicitly stated otherwise). This is only the case with the next function. When using, for example, only *init(a):=1*, the value of a is only defined at time 0 and it is *not* defined at time 1 and further.

In this model a is allways 0:

0:

```
a := 0;
```

because at all points in time *a:=0*, according to the model.

The values of a over time will be the natural numbers in the following model:

$0, 1, 2, \ldots$:

```
init(a):=0;
next(a):=a+1;
```

There is *no* integer of which it is *im*possible for a to have its value in this model:

$0, 1, 2, \ldots$ *?:*

```
next(a):=a+1;
```

The reason for this is that the next value of a is defined in terms of the undefined current value of a.

2.5.3 The SMV language

SMV is also strong in modelling synchronization problems. This, however, is not what we want to model, so we will leave synchronization and some other language elements out of this introduction.

In SMV variables are declared using a semicolon. Furthermore, types can be defined as ranges or sets.

Variables:

```
b : boolean;
b2 : {0,1};
c : 0..10;
d : {Monday, Tuesday, Wednesday, Thursday, Friday};
```

To these variables one can assign values. Note that this involves the time in which the "statement" holds.

Variables 2:

```
b : boolean;
b2 : {0,1};
c : 0..10;
d : {Monday, Tuesday, Wednesday, Thursday, Friday};

b := 1; /*set b to true*/
b2 := 0; /*set b2 to false*/
if(b & b2)
{
        next(d):=Monday;
}
else
{
        c := 2;
}
```

If a certain constant value or type is used a lot, it is possible to define them in a more readable way. Constant values can be defined, much like C, such as "#define MAXBYTE 8". Types can also be defined such as "typedef BYTE 0..MAXBYTE;"

It is possible to use conditional blocks with the *if* statement, as can be seen in the example above. Also it is possible to use operators like the following ([McM01]).

!	logical *not*
&	logical *and*
|	logical *or*
−>	logical *implication*
<−>	logical *equivalence*
=	logical *equality*
!=	logical *disequality*
<	logical *less than*
>	logical *greater than*
...	

One of the first things to get used to is the effect of the discrete time line. The following code might seem correct SMV but it is not:

Variables 3:

```
c : 0..10;
d : {Monday, Tuesday, Wednesday, Thursday, Friday};
init(c) := 0;
if(c=0)
{
        d:=Monday;
}
else
{
        next(d) := Tuesday;
}
```

The reason for this is that c might be defined initially as 0, it is not stated in this model what c is from time 1 on. That means that this is possible:

time	c	next(c)	d	next(d)
0	0	undefined	Monday	undefined
1	(1)	undefined	undefined	*Tuesday*
2	(0)	undefined	*Monday*	

In this table the values of c between brackets are assumed for the counter example, which is allowed since the value is not defined to be something else.

In this way, at time 1, d is already stated to be Tuesday for the time 2, but then it is also assigned to be Monday at time 2 itself. This is not allowed in SMV and the tool will not accept this as a valid model.

Another usefull statement in the SMV language is the *default* statement.

Default:

```
b : boolean;

default
{
        b:=1;
}
in
{
        ...
        b := ...
        ...
}
```

This allows you to state that some assignments to variables must hold by default, that is, *unless* they are assigned in the *in* part of the statement. In this way they do not contradict. The default statement is merely a convenience notation since it is also possible to state all of this explicitly in the *in* part.

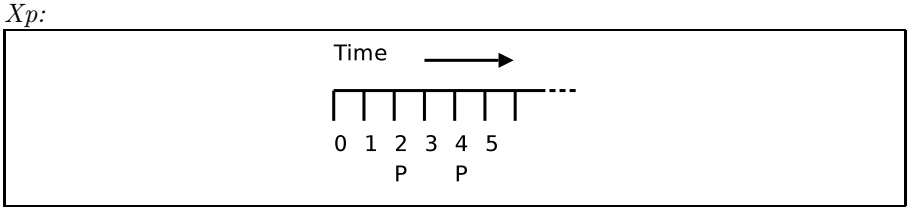
2.5.4 Properties

Using the same variables and time definitions it is possible to state properties of the model. The SMV model checker will then attempt to verify these properties.

To handle time in the properties the following functions can be used [McM99]:

- X

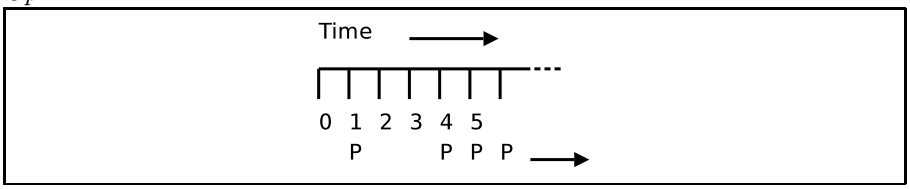
Xp:



X indicates the next time. Xp is therefore true at a certain time if at the next point in time p holds. In the figure Xp holds at time 1 and 3 only.

- G

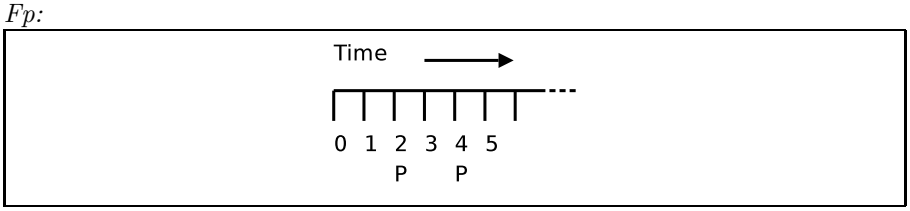
Gp:



G stands for globally. Gp is true at a certain time if from that time on p will hold. In the figure Gp holds at time 4, 5 and further. Gp does of course not hold at any time before 4 since at time 3 p does not hold.

- F

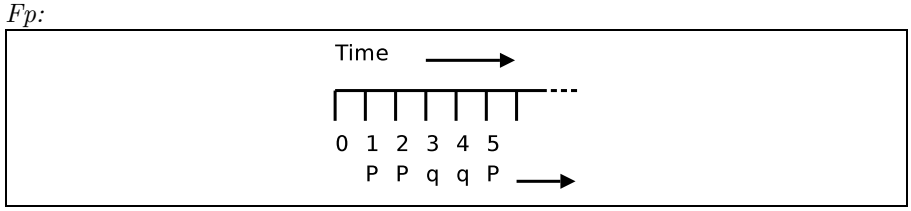
Fp:



F stands for future. Fp is true at a certain time if from that time on there is or will be a time at which p will hold. This means in the figure Fp holds at time 0, 1, 2, 3 and 4.

- U

Fp:



U stands for until. pUq is true at a certain time if from that time on p will hold at least until q will hold. This means in the figure pUq holds at time 1, 2, 3 and 4.

Of course there are different paths of states when using nondeterminism. Xp, Gp, pUq will hold if the description of it above holds for all paths. Fp will hold if the description will hold for all possible current states and at least one path.[5]

A typical example is safeness, where one could state "Gp" indicating p should always hold. Fairness could be stated as GFp.

These functions now allow us to construct properties over time and add them to the model.

In this work we only use the main module of an SMV model which has the following straightforward structure:

Main:

```
module main()
{
        ...
}
```

A simple and complete SMV model could be the following:

[5]We have tested the paths required with the SMV implementation described in section 3.7

Main:

```
module main()
{
        a : -100..100;

        init(a):=0;

        if(a < 50)
        {
                next(a) := a + 1;
        }

        /*
                always a is greater than
                or equal to zero
        */
        greaterOrEqualZero : assert G (a>=0);
        /*
                always from some point on
                a will remain the same
        */
        maximum : assert G (F (G(a = (X a))));
}
```

Both of these properties hold for this model.

Chapter 3

Fault injection models from Java code

In this chapter we start out with an overview of some of the problems in translation, then we continue by describing the translation of different parts of the Java language into the SMV language. After this we describe how the attack on smart cards can be modelled on top of that translation.

We examine the experimental SMV generator and how to use it, after which we sum up some other problems that had to be solved, but are less important for the overall ideas behind the generation.

In section 3.7 we provide some details such as versions of software and hardware we used for those who are interested.

3.1 Overview

Data types The Java data types differ from those in SMV. The most important differences are that Java types can be more dynamic (references) and modelling the full range of Java data types (this is not necessary in most cases), will easily cause a state space explosion.

Names and hierarchy Java has a hierarchical structure of contexts in which names might exist (namespace). In SMV there is no such hierarchy. All local names used in the Java code have to be translated into some global name, by which a human reader must still be able to find the original Java variable.

Sequential code In an SMV model there is no sequentiality. In Java there is, of course. Translating this can be done, but has quite some consequences and difficulties.

Method structure Java uses methods, a standard structure by which code can be re-used. Jumping back and forth to different parts of code (which happens in another way with *if* and *for* statements, for example) is not straightforward.

Expressions Java expressions can have (deep) side effects which have to be taken into account. Expressions therefore have to be flattened and evaluated in multiple steps.

Attacks Apart from the Java code also the attacks must be modelled. The way this is done can have major consequences for the rest of the model and state space. Preferably the Java code and attacks are modelled separately to keep the generated model as clear as possible.

3.2 Modeling Java in SMV

We want to generate an SMV model from Java code. This model should then allow us to find vulnerabilities of the code to certain modelled attacks.

We think that the starting point of the translation should be to translate the Java into SMV by the statement as much as possible and that we should try to keep the original structures of the Java code similar if we can. It will make the model easier to read (and adapt) for human beings.

Java		SMV
		module main()
		{
		declarations
statement 1	→	translation of statement 1
statement 2	→	translation of statement 2
statement 3	→	translation of statement 3
...		...
		}

In the next section we will explain how we translated different elements of the Java language into the SMV language.

3.2.1 Data types

First of all we will look at variables and their values. This is one of the easier steps in the translation. There are different data types in Java, which all, to some extends, can be translated into SMV data types. Data types in Java can be referenced and primitive [JG05]. All primitive types are listed below:

- booleans
- byte, short, int, char, (long)
- (float), (double)

Those between brackets do not currently exist on Java cards.

The reference values are pointers to a *class instance* or *array* objects, and a special null reference, which refers to no object.

In SMV the following data types exist:

- boolean, enumerated and subrange types
- arrays
- structures

Together these form a basis for translation.

One has to take into account that usually not all of the possible values of a variable are required. For example the commonly used loop variable as in the following example will usually not have other than $0 \ldots 10$ for example.

Simple loop:

```
for (int i = 0; i < 10; i++)
        printSomething(i);
```

This could be significantly important in reducing the state space. Instead of the range $-2,147,483,648 \cdots +2,147,483,647$ one only has to deal with 10 different values.

With the subrange data type it is possible to translate bytes, shorts, integers, longs and chars.

Also it is possible to translate floating point numbers in this way, taking into account some maximum precision, which exists in most implementations of such primitives, including Java. For example a floating point number with a maximum precision of two digits after the dot can be translated to

an integer by multiplying it with 100. At this moment doubles and floating point numbers are not supported by Java cards so it is not relevant (yet).

With these steps it is possible to, quite generally, translate Java primitive variables into SMV code. The struct data type of SMV allows also class instances to be translated. However, since these are not commonly used in the low-level smart card programs, we did not look into that further.

3.2.2 Arrays

Both Java and SMV support the use of arrays and, more specifically, SMV supports arrays in a way that Java arrays can be translated without much trouble[1], even though the definition and use of arrays might be different.

Suppose we have the following Java code:

Java array:

```
int a [] = new int [10];
a[0] = 2;
```

then we can translate this into

SMV array:

```
a : array 0..9 of MININT..MAXINT;
a[0] := 2;
```

which is quite easy.

Although they do not exist on Java cards, also more complicated arrays, such as multidimensional arrays, are no problem since SMV allows for the same constructions.

Java array:

```
int b [] = new int [10][2];
b[0] = 2;
```

We can translate this into

SMV array:

```
b : array 0..9 of array 0..1 of MININT..MAXINT;
b[0] := 2;
```

which is also quite easy.

[1] Although assignments are tricky (section 3.6.6)

3.2.3 Names and hierarchy

The hierarchical structure of Java, which also becomes apparent with the use of names in Java (namespace), is something that has to be translated to the "flat" structure of an SMV model. We can try to translate the names of methods and variables in a way that we can read the place in the Java hierarchy from the SMV variable name.

There are more ways to translate code. Often the names are one of the first things to "disappear" in the translation process. In fact they are usually translated into some unique number or address. A processor won't mind.

Since the SMV code is not only meant to be read by the SMV tool but also by humans, so they can check it or modify it manually, it is important that the names are kept or translated in a way a human would be able to figure out what is going on. Having stated that, please consider the following code:

Names:

```
public class a
{
        int a;
        public a()
        {
                int a = 0;
        }
        void a(int a)
        {
                if(a==0)
                {
                        int b = 0;
                        a = 2;
                }
                else
                {
                        int b = 2;
                }
        }
}
```

Obviously this code does not make much sense, but it is a correct Java class according to the Java language specification. It is clear that we have a lot of a's and b's that often do not always refer to the same instances of variables or methods.

In order to overcome problems with these situations we decided to tackle

this thouroughly. Of course we could have stated that the input Java file has to be adapted so this problem will not occur but that seemed really inconvenient for the developer that would use the Java to SMV translator.

Visibles

We called the construction we used "Visibles", since it contained information about the visible names and the variables or other elements to which the name would belong at a certain time. Visibles is the name of a single structure. A more elaborating name would solve the plural/singular confusion we will now cause, but since we use the word often, a long name is also inconvenient. The basic idea is as follows anyway:

A name refers to a different element depending on the point in the code where the name is used. Call the area where the name is valid a *block*. Then one can define a structure *Visibles* where for each block a new Visibles is created, which can contain names and a reference to the surrounding block. If one searches the element belonging to some name one can find it by looking at the Visibles of the statement currently looked at. The Visibles can check whether the name exists in the current block. If so, the element is returned. If not the Visibles forwards the request to the surrounding Visibles of the surrounding block. Also you can define in this way which of the elements (variables, methods etc) take preference, if more than one of them exist in one Visibles.

In the following table we try to clarify this:

line nr	*code*	*Visibles nr*	*names*	*parent*
1	int a = 0;	0	a(1)	
2	void a ()	0	a(1),a(2)	
3	{	1		0
4	int a = 2;	1	a(4)	0
5	if(a==2)	1	a(4)	0
6	a = 0;	2		1
7	}	1	a(4)	0

The name "a" belonging to the element declared at line 4 is indicated as "a(4)". The SMV generator reads through the statements one at the time and therefore the list of names can grow during the generation. Visibles number 1 at line 3 of the process is still empty. Then, when it reaches the line "int a = 2", it will remember that variable as belonging to the name "a".

At line 6 there is an assignment of "0" to the variable "a". The Visibles at

that time (Visibles 2) does not contain the name "a" and therefore forwards the request to its parent, which is number 1. That Visibles has the name "a" registered and therefore at line 6 it is clear the variable referred to is the one declared at line 4.

How exactly the Visibles should behave, where a new Visibles is used and a Visibles ends can be found by looking closely at the Java documentation [JG05]. Whether or not the example of Visibles in the table above is correct according to the Java specification, this technique is perfectly capable of solving the problem of the many equal names.

Global Variables

In the implementation of the generator we made, we also used the idea of Visibles idea to solve the following problem: In SMV only "global" variables exist. SMV only allows for variables that exist at all times where Java allows for local variables. We solved this problem by generating a unique name for each variable in the following way:

We provided each *Visible* with an unique (generated) name so we could give each variable a unique prefix depending on where it is created (in which Visibles). The outermost Visibles, the root Visibles, contains the Java class's global variable (field) names and method names. It is called "". Then the Visibles of which the root is parent are the outer Visibles of each of the methods (including constructors). These Visibles can have the name of the method because this makes it easier for humans to see what part of the code a variable belongs to. Inside the methods there were no names we thought of as usefull and we decided it would be good enough to number each Visibles with an ascending unique number retrieved from the parent Visibles. We used a separator "_" to put it all together. Therefore the most complicated names would have this form:

$$name = methodName\_VisiblesIndex\_\ldots\_VisiblesIndex\_variableName$$

For example a very complicated name $printArray\_0\_1\_i$ would indicate the variable i created in the second Visibles of the first Visibles of the "printArray" Visibles.

In this way all local variables can be created as global variables. A note here: for readability it is not a good thing, but for performance it will be useful to reuse the local variables where possible. If two methods, that can never (indirectly) call eachother, both use a local integer variable, it saves state space to have them use the same variable in the model. Now it is

also possible to do this, but the author of the Java code has to use a global variable in Java already and figure out the dependencies manually.

3.2.4 Sequential code

The first interesting difference between Java and SMV we came across that had to be solved was that Java is based on some defined order of the operations. SMV does not allow such a thing directly. In SMV every statement must be true at all times and no contradictions may occur in the model at any time.

For example the following Java code

Java example:

```
boolean a;
a = 0;
a = 1;
```

can not be translated to SMV code like the following, since it would be a contradicting model:

Contradicting SMV snippet:

```
a : boolean;
a := 0;
a := 1;
```

It contradicts because a cannot have the values 0 and 1 at the same time.

The way we modelled this is by introducing a counter. With each operation in Java the counter increases in SMV. Because this counter has only a single value at any point in time, the counter allows for one operation to "happen" at a time. The above snippet of Java would therefore translate to something similar to

Non-contradicting SMV snippet:

```
a : boolean;
counter : 0..MAXINT;
init(counter) := 0;
if(counter=0){
        a := 0;
        next(counter):=1;
}else if(counter=1){
        a := 1;
}
```

Also more complicated code will be generated nicely using this method, preventing ambiguities in the model that did not exist in the Java code:

Java snippet:

```
a = 0;
a = 1;
b = a;
```

SMV snippet:

```
init(counter):=0;
if(counter=0)
        next(a):=0; //ensuring that at the ''execution''
        //of the java line a=1, a will be 0
        next(counter):=1;
else if (counter=1)
        next(a):=1;
        next(counter):=2;
else if (counter=2)
        next(b):=a;
        next(counter):=3;
```

In this way it is possible to model an ordered language like Java in a non-ordered language like SMV

As a result of this, the structure of the generated model changes a little:

Java		SMV
		module main()
		{
		declarations
		now the conditional part:
statement 1	→	if (...) translation of statement 1
statement 2	→	else if (...) translation of statement 2
statement 3	→	else if (...) translation of statement 3
...		...
		}

3.2.5 Method structure

The second important thing to model was the idea of Java methods. This can be accomplished in many ways. Two of the extremes are:

1. Model methods in the same way as it is implemented (and executed) in Java. That means there is a piece of code you can *jump* to and

come back from. (Using a stack)

2. *Insert* the methods code at all points where the method is called.

Of course hybrid solutions are also possible. There are additional problems with all solutions. For these two extremes the main problems would probably be:

1. *jump*: Modelling jumping is not straightforward and neither is modelling Java's parameter handling.

2. *insert*: This method would give problems in that the model would grow in size enormously, depending on the original Java code.

Furthermore the *jumping* would allow for the modelling of *loops* and would be convenient to do anyway. That is, if the parameter handling can be modelled in SMV. In the generator the jumping solution is implemented.

Parameter handling

The following example shows how decent handling of parameters is vital in modelling the method structure. There did not seem to be a shortcut so we decided to work it out thouroughly.

Consider the following code:

Java snippet: parameters:

```
public int giveNumber(int a, int b)
{
        return a+b;
}
public void main()
{
        int c = 0;
        if(giveNumber(2, c)==giveNumber(c, c))
                c = giveNumber(c, c);
}
```

The way we decided to translate Java parameters into SMV is to create a "local" variable for each one in the method. Then, at the call of the method, the values of the parameters are copied to those variables after which the method can actually be "executed". In this way all Java card parameter usage can be handled correctly[2].

[2]Arrays will be discussed next and other referenced types can be handled in a similar way

Array parameter Arrays as parameters provide a difficulty since the copying of arrays is not the same for Java and SMV as will be explained in section 3.6.6. One easy way to handle arrays is to copy the arrays and, after the method has been called, copy them back. This solves the most common situations of the referenced arrays. However a problem could occur if *the same* array is passed multiple times as parameter and the parameters influence the outcome of the called method. For example in this nonsense code where, when called with the same array twice, the value of array[0] will be 1:

Java snippet: parameters:

```
public void assign(int [] a, int [] b)
{
        b[0] = 0;
        a[0] = 1;
}
```

When copying the values back, in this case starting with a, the trick won't work. For the purpose of this research we did not go further into this. This situation is not likely to occur often in elementary smart card code and especially it did not occur in the examples we investigated.

Jumping

Since parameter handling worked out we could use jumping as a way to translate methods. We use the fact that on a Java card only one method can be active at a time. Therefore using a variable, indicating which method is active, and using line number ("process counter like") variables for each method, it is possible to keep track of exactly where we are (about to go) in the execution. We called this variable "active" since it indicates which method is active. Methods can then call each other by setting *active* to the method to be called.

Since methods also have to return to the callee we must keep track of who called who. This is easy if there is no recursion, since we can add another variable to each method indicating who called it. In this way all other calls are allowed. We named the variable "methodName_returnTo". The following example illustrates the whole idea:

Calling Java:

```
public void a()
{
}
public void main()
{
        a();
}
```

Calling SMV:

```
a_CL : 0;                   /*CL = current line*/
a_returnTo : {main};
main_CL : 0..1;
active : {a, main}
init{active} := main;
init{a_CL} := 0;
init{main_CL} := 0;
if(active = a)
{
        if(a_CL=0)          /*returns immediately*/
        {
                next(active):=a_returnTo;
                next(a_CL):=0;  /*reset the CL!*/
        }
}
else if (active = main)
{
        if(main_CL=0)       /*call a*/
        {
                next(active) := a;
                next(a_returnTo) := main;
                next(main_CL) := 1;
        }
        else if (main_CL=1)     /*end program*/
        {
                /*end, for example, by not resetting CL*/
        }
}
```

It becomes clear that the structure is now augmented with special cases of the conditional statements:

Java		SMV
		module main()
		{
		declarations
method 1	→	if(active=method1)
...	→	if (...) ..
...	→	else if (...) ..
method 2	→	else if(active=method2)
...	→	if (...) ..
...	→	else if (...) ..
...		...
		}

Now, using this structure, the parameter handling can be added, as well as the return values, without much trouble in the same way. An example of this will follow later on. Still there do arise some problems. Consider the following code:

Calling Java 2:

```
public int a(int a)
{
        return a + 1;
}
public void main()
{
        int c = a(2);
}
```

At first the problem might not be so obvious. Parameter copying, for example, is not really hard. If during the SMV generation the methods are known and therefore the required parameters and their names are known, it is easy to add SMV code for each method call which copies the parameters.

A multipass generation would work fine for this. In a multipass generation the code is read through multiple times instead of just once. This often prevents the lack of information during the generation because information is simply not yet read. The following example illustrates the difference:

Single pass would not work:

```
public void b()
{
        a(2);
}
public void a(int a)
{
}
```

In a single pass it is not possible to know, in advance, what the parameters of a will look like. When generating the code for b however, that is required information. In a multipass generation it is easy to collect, for example, method information, such as parameters, so it can be used later on in the second pass.

For the experimental generator we wrote we did use a single pass generation because recursion is not allowed and the code is not so complex that it requires a multipass generation. Worst case scenario for common code would be that a method has to be copy-pasted to another part of the file which is easy and has to be done only once. It is possible and it is not hard to implement a multipass generator if that is required.

Note that even though recursion is not possible, still it is possible to write code that requires a multipass generation:

No recursion during execution but single pass would not work:

```
boolean firstAthenB = ...;
public void b()
{
        if(! firstAthenB)
                a(2);
}
public void a(int a)
{
        if(firstAthenB)
                b();
}
```

Possibly other examples can be found and perhaps it might depend on your definition of recursion.

Continuing on the Java example on page 45. Obviously the return value of the method a must be known prior to assigning it to c. This means that the elements of an *expression* must be handled in some order. This is perhaps not a surprise since in most, if not all, languages this is the case. However since calling the method also takes multiple steps the model that

is generated will contain extra steps for all the “administrative work”. More on the topic of splitting an expression up in multiple manageable parts can be found in section 3.2.6 where it is required for accurate modelling.

The generated code would look similar to the following:

Calling SMV 2:

```
a_CL : 0..0;                    /*CL = current line*/
a_returnTo : {main};
a_value : MININT..MAXINT;/*return value of the method*/
a_a : MININT..MAXINT;           /*parameter called a*/
main_CL : 0..2;
active : {a, main}
main_c : MININT..MAXINT;
init{active} := main;
init{a_CL} := 0;
init{main_CL} := 0;
if(active = a)
{
        if(a_CL=0)              /*returns immediately*/
        {
                next(active):=a_returnTo;
                next(a_CL):=0;          /*reset the CL*/
                                /*set return value*/
                next(a_value):= a_a + 1;
        }
}
else if (active = main)
{
        if(main_CL=0)   /*call a*/
        {
                next(active) := a;
                next(a_returnTo) := main;
                next(main_CL) := 1;
                next(a_a) := 2; /*set parameter*/
        }
        else if (main_CL=1)     /*returned from a*/
        {
                next(main_c) := a_value;

        }
        else if (main_CL=2)
        {
                /*end, for example by not resetting CL*/
        }
}
```

3.2.6 Expressions

Expressions in Java can be complicated and can involve a lot of calculations that have to be done separately in SMV. We must therefore also translate these expressions into multiple SMV statements. The following Java can not be translated without care:

Complex expression:

```
int [][] big;
public int [] doSomething(int a [])
{
        if(a[0]==0)
                return a;
        return big[1];
}
public void a()
{
        /*and now the complex statement expression:*/
        big[0][0] = (doSomething(
                        big[doSomething(big[0])[2]]
                        )[2] = 2);
}
```

The first important thing is that in this complex line there is actually an order in which things happen. The calculation of the result of the expression consists of multiple steps where each next step depends on the *result* of the current step.

For the generation of SMV code we applied this as well. Because of the possibility of one line of Java having to consist of multiple SMV lines (if only for method calls) we had to work out a way to handle this correctly.

We wrote a method for each expression type which resulted in an ExpressionResult object, which would contain information on the result of the expression. The result could be a variable in SMV or a value.

Apart from the actual result of the expression, the generator could also generate SMV lines that had to be "executed" before the result would be valid. A typical example would be a method call where first the lines would be generated for the method call itself and then the ExpressionResult containing the result variable of the method would be returned for further processing.

In this way it was possible to generate the Java expressions correctly in SMV.

The way these "blocks" of SMV code, that have to precede others, are

handled is dealt with in section 3.2.7.

3.2.7 Blocks of SMV and jumping

SMVBlocks

One of the important conclusions we came to was that some pieces of Java could be generated into SMV statements that could all be "executed" at the same time.[3]. On the other hand some statements must precede others. The next example shows Java code that requires only a single "block" of SMV statements.

One SMV line Java:

```
...
a = 0;
b = 2;
c = 2+5;
d = e+d;
...
```

To make sure these kind of Java segments would not generate too many current line values, we implemented the idea of an SMVBlock. This SMVBlock could contain many SMV lines that would be used at a *single* value of the current line variables (the method's current line variable and the active variable).

One SMVBlock:

```
if(currentLine=...)
{
        next(a) := 0;
        next(b) := 2;
        next(c) := 2+5;
        next(d) := e+d;
}
```

This concept can be used to some extremes reducing the state space, which is one of the most important problems in model checking. Even Java statements might be swapped in order to reduce the maximum current line as in the following example, where the middle two lines can be swapped to reduce the state space from three SMVBlocks to two SMVBlocks due to the (in)dependencies of the Java statements.

[3]In fact it is not executed at a point in time but must be true in the case the current line and active variable points at those lines

Swappable Java:

```
a = 0;
b = a;
c = 0;
d = c;
```

We did not implement this to such extremes but we want to note that implementing this carefully could lead to better results than our experimental implementation.

SMVBlocks are stored in order per method, so the current line counter of the method will simply point at one of these blocks, going over them one at the time, in the same order.

There is an important note to SMVBlocks. If we model attacks that reset the device, the actual reset will occur right after the SMVBlock. Consider the following example:

Interrupt:

```
a = 0;
b = 0;
```

Translated into a single SMVBlock, a reset attack on the variable a would not prevent $a = 0$ from happening. On the actual smart card it would. If the assignment to b does not change the behavior of the reset, there is no problem. This, however, is not always the case. SMVBlocks *can* therefore be very convenient but have to be used with care.

Local and global SMV

In SMV some code must be global, such as the definitions of variables and assigning initial values to them. Other code must be local, inside if-else statements, in order to avoid contradictions in the model (as discussed in section 3.2.4).

In generating the SMV we explicitly implemented ways of adding SMV to global or local lines (Local lines are implemented in SMVBlocks where global SMV does not require line numbers and things alike).

Also in the global SMV it is possible to set all variables to their, or some (if undefined), initial value. If an integer variable initially has no value it would give MININT to MAXINT possibilities to start with, while possibly the value is not important. Some modelcheckers might be able to figure this out, but to be sure we implemented it this way. Setting it to some default value reduces the initial states and thereby the state space.

General correct SMVBlock jumping

One SMVBlock corresponds with one value of the active variable and the current line value of the corresponding method. Therefore jumping would go by the block. This worked out quite convenient as we will explain now.

The parse tree, generated by the adapted Jabstract (section 2.4) tool, is of course read through starting from the root. The SMV must largely be generated starting at the leafs of the tree, which is exactly the other way around. Now the convenient idea we used is that after a child is completely parsed (and the SMV of it is generated or prepared for generation), it will not generate any more SMV lines. Therefore if you *first* generate the SMV of the *child* and *then* the SMV of the *parent* you have all the information you need to jump around correctly. The next paragraphs will explain how this is used in the generation of the SMV of some jumping statements such as loops.

3.2.8 Examples of statements

If-then(-else)

A nice example of a Java statement to start with is the if statement which looks like this:

if Statement:

```
if (condition)
{
        true case
}
else
{
        else case
}
continue
```

In this statement the *else* and therefore *else case* part is optional. The condition is an expression which has a boolean value. The true case is a block as discussed in section 3.2.3 on page 38 as is the else case part. The blocks might consist of a list of statements.

Generating SMV code for the if statement can be done in the following way, building on the ideas as discussed (and implemented in the SMV generator) earlier.

First of all the condition part of the if statement must be translated to SMV. It might generate multiple SMVBlocks. That is not a problem. When

the SMV is generated we have the result of it as a piece of SMV expression, as discussed in section 3.2.6. Having the generated SMVBlocks and that line we can now picture the jump:

if Statement condition filled out:

```
generated SMV of condition
if (piece of SMV line returned from generation)
{
        true case
}
else
{
        else case
}
continue
```

Now since we use line numbers this won't work out with the curled brackets as shown in the next example:

if Statement condition filled out line numbered:

```
if(line number = ...)
{
        generated SMV of condition
}
else if (line number = ...)
{
        if (piece of SMV line returned)
        {  /*this is where the brackets will not work*/
}
else if (line number = ...)
{
                true case
}
else if (line number = ...)
{
        }
        else
        {
...
```

The thing we came up with to solve this was sorting out all the jumps in advance and then setting up the SMV if statement.

We generate the condition SMV and the true case SMV (and the else case SMV if it exists) and we used the principle that *after the generation of*

an SMVBlock no SMVBlocks will be inserted before it. Therefore we could figure out the line numbers belonging to the SMVBlocks and had all the information required to set up the jumps.

Now we did the following to set it all up:

After we generated the condition SMV we added a new SMVBlock which was empty. Later on we would fill it with the SMV code for the jumping. Then we generated the true case SMVBlocks. At that point we knew the SMVBlock line number *after* the true case where either you would want to continue or jump to depending on the existence of the else case.

if Statement Blocks:

```
condition SMVBlocks
        add new block where we can set up the jump
true case SMVBlocks
        if the else part exists, jump from
        here to after the else case
        or remember it as the continue block
else case SMVBlock if it exists
        remember this as the continue
        block if the else case exists
```

The SMVCode then would look like the following (in the case the else part exists):

if Statement SMV:

```
generated SMV of condition
else if (line number = ...)
{
        if (piece of SMV line returned)
        {
                next(line number) := trueCaseStart;
        }
        else
        {
                next(line number) := elseCaseStart;
        }
}
else if (line number = trueCaseStart)
...rest of true case SMVBlocks
else if (line number = lastLineOfTrueCase)
{       /*added to jump over the else case*/
        next(current line) := continuStart;
}
else if (line number = elseCaseStart)
...rest of else case SMVBlocks
else if (line number = continuStart)
...the rest of the model
```

In this way we can generate SMV for the if statement. It does not require much more state space as the number of variables do not increase and only a few SMVBlocks have to be added.

For-loop

The for loop is similar to the if statement when it comes to the jumping. The statement looks as follows:

for Statement:

```
for(init;condition;update)
{
        statements
}
continue
```

Where it is executed in the following way, translated to the probably even more familiar while loop:

for statement as a while statement:

```
init
while(condition)
{
        statements
        update
}
continue
```

which is the way to generate it.

The SMVBlocks for the init case can simply be generated. Then the conditional blocks have to be generated with more care as they consist of the blocks and the piece of SMV line that has the value of the condition. The condition probably has to be evaluated multiple times, so the first SMVBlock of it has to be remembered to jump to later on.

After the generation of the condition blocks the jumping block can again be added after which the statements blocks can be generated as well as the update block(s). Now we can add a block to jump to the first condition block to re-evaluate it.

Now we know what block number the continue block will have (which is not necessarily a separate block but simply the next block that does not belong to the for loop). We can set up all the jumping now by adding the following SMV lines to the SMVBlock we reserved for jumping earlier on, to have it all look like this.

for statement as a while statement:

```
init
        remember the next block as the
        first condition block
condition blocks
if(current line = ...)
{
        if(condition piece of SMV line)
        {
                next(current line) := current line + 1;
        }
        else /*terminate loop*/
        {
                next(current line) := continue line;
        }
}
statements blocks
update blocks
continue...
```

3.3 Modelling attacks

The attacks that this research is about can be summarized by attacks that make a read operation fail or a write operation fail. Also each attack could reset the device or make it continue, even though the operation failed.

Each attack would fit in a single field in the following table:

	Read	Write
Continue	RC	WC
Reset	RR	WR

Note that the RR attack will in fact only reset the device, since the value read is not used before the device is reset. A failed operation will result in any of the values in the range of the type of the variable, instead of just the intended one. Modelling the attacks as stated here gives a clear perspective on the possibilities of attack, weaknesses of code and ways to set these attacks up in a modelchecker. Also it is interesting that, although attacks are likely, some combinations are not realistic. For example it might be considered unrealistic to have an unlimited number of write continue attacks at exactly all write operations in a piece of code. Therefore *counting* the attacks per category that have been used is convenient. This can be

done either in the model explicitly, requiring more state space, or it can be done by setting up the property in a more complicated way, requiring more of the user. This can also be automated to some extend, so the user will not necessarily have to do that by hand.

Apart from these differences, we can also distinguish between the different elements of the hardware that are under attack. In the Java card we have two different kinds of memory typically used to store variables: the EEPROM and RAM memory. Usually the RAM memory is used for temporary variables, such as local variables, while the EEPROM memory is used for variables that have to be stored for a longer period of time, such as global variables (fields). These different kinds of memory require different techniques to attack. Therefore it is useful to distinguish between these (and possibly other) kinds of memory. In our approach we keep track of the attacks per memory type.

We will now look at the way to model failing read or write statements, how to count these and finally how to reset the device, in case of a reset attack. We will replace the read and write statements by statements that use in-time updated special variables. You could say we wrap "get and set variable pairs" around each read and write statement, allowing the get variable to be different from the set variable of the same pair. This is not the only way to accomplish this as we will explain. The attacks, as stated in the table, can be modelled by replacing each read or write operation by an operation that allows a registered error to occur. On top of the regular generation of SMV the model for these attacks has to be added. This model can be constructed in at least two ways. The statement below will serve as an example.

Java read write example:

```
a = b + 1;
```

1. Adding read or write "methods"
   Similar to the way a common Java method is translated to SMV it is possible to create *read* and *write* methods to loop values through the attack model. Then a statement like the above could look like this:

   Java read write example:

   ```
   write(a, read(b) + 1);
   ```

   The downside of this is that it, for each and every read or write opera-

tion, will cost some values of the current line counter thereby increasing the state space a lot, since a lot of reading and writing operations will have to be modeled. Also it will give some administrative work on copying all variables back and forth. For humans it is probably worst that the code becomes very unclear because of all the added code in the model throughout each modeled statement.

2. In time replacement
   Another solution is in time replacement where we use the fact that in SMV also in current time we can do operations on variables, contrary to the *next* operation we used so far. We have thought about doing this and came up with a possibility to do the operations in time. The idea is somewhat like this:

   Java read write example:

```
setread = b;
setwrite = getread+1;
a = getwrite
```

   In SMV it will not look much more complex so it is still somewhat readable and understandable. The reading and writing operations under attack are modeled via the get and set variables, which are respectively the output and input variables of the operations. Obviously adding variables means increasing the state space which is not what we aim for but we figured we could still give it a try.

3.3.1 In time attacks

For reasons of readability, state space, and having the added code not throughout the model but all put together, leaving the Java code specific piece more intact, we chose to try the solution which we will now further describe.

The idea is that, in order to avoid waiting for the attacks to be performed by means of jumping to another modelled method or by putting the entire Java specific part in a $if(attackReady)$ clause, it is better to model it in a way that before the values are assigned by the *next* operator, thus in current time, the attacks are simulated. Since SMV allows also you to set variables at current time this could be worked out like this:

For each attack to be modeled we could create a get and set pair of variables where the set could be used as input of a value and the get could

be used to retrieve a value. This last one will be either the modified variable, if an attack took place and modified it, or it could be the value of the get variable (which is the original variable's value).

Tossing a coin in SMV is easy. Allow a variable to have a value from a set. It can only have one value of course, but SMV will check all options in the model checking process. This means that tossing a coin is as simple as

Attacking:

```
boolean attack;
attack := {1, 0}
```

where 1 is true and 0 is false.

Now if the attack variable has the true value we allow the get variable to have any possible value and otherwise we simply copy the value:

Get set:

```
boolean attack;
attack := {1, 0}
if(attack)
{
        get = {MININT..MAXINT};/*allow all*/
}
else
{
        get = set;
}
```

We can use this for reading as well as writing. Up until now it is quite straightforward but this won't do in practice yet.

First of all, if we want to translate expressions directly, we also have to deal with these kind of expression:

Multiple read write operations:

```
a = b + c + x - methodA() * (x = methodA() + 2);
```

These are some of the reasons this is complicated:

- Multiple variables are read at a single point in time in the generated SMV, a single SMV Block.

- Some variables (fields like perhaps x here) might be stored in a different kind of memory which possibly requires a different (kind of) attack and therefore administration.

- A method might return a different value each time it is called (e.g. a random number method), yet be used multiple times in an expression, even if the parameters are the same.

- Even multiple write operations are possible, also to different kinds of memory (x).

- Fixed primaries $(1, 2, 3, \ldots, a, b, \$, ", null, \ldots)$ might also be read from (possibly faulted) memory.

The multiple reading or writing is not hard to solve but there are more ways to do it. One way is to have an array of get and set variables. Also the difference between the memory types can be added here as different get or set pairs.

Multiple read write operations SMV:

```
readerset : array {0..reads at a time of a memtype}
        of array {0..memtypes} of integer;
readerget : array {0..reads at a time of a memtype}
        of array {0..memtypes} of integer;
writerset : array {0..reads at a time of a memtype}
        of array {0..memtypes} of integer;
writerget : array {0..reads at a time of a memtype}
        of array {0..memtypes} of integer;
```

These SMV variables will allow for multiple read and write operations at a time. This would solve most of the problems stated above. The fixed primitives can of course be "wrapped" by some get and set pair, so these can be modelled in the same way.

Of course it is not brilliant to have these variables set up as an array since some variables might not be used. These would make the model unnecessary complex.

During the generation it is possible to find out how many reading and writing operations are used for each memory type at a time and this will allow the complete setup so far. We did this by having a ticket system for each SMV Block, which would provide the first unused index for each read or write operation, depending on the memory type. After the statement is completely generated, the ticket system is reset. It also kept track of the maximum numbers so at the end the get and set arrays could be generated.

In future time

Now we do face a far more serious problem with the method calls since these will have to be calculated over multiple line counter values. The problem we came across, at least here it is a problem, is that assigned values were not remembered:

SMV "forgets":

```
counter : integer;
init(counter) := 0;
value : integer;
if(counter=0)
{
        value := 1;
        next(counter) := 1;
}
else if (counter=1)
{
        ...
}
```

Obviously when counter equals 0 the value of *value* is 1, however at the next point in time where counter equals 1, the value is undefined. SMV does not automatically remember the value. How we managed to solve this we will explain now.

In order to use the variable again in the next SMV time step it had to be remembered. So what we did was using a separate variable to remember the value. This increases the state space quite a bit, but it seemed worth at least trying it. Using this we can add the statements similar to

SMV default read:

```
next(defaultread[i][j]):=getread[i][j];
next(defaultwrite[k][l]):=getwrite[k][l];
```

The first index of the array indicates the number of the pair. The second index indicates the memory type.

Mind that we need to use getread here since the modified value has to be remembered instead of the original value. Modification is only done once to keep it countable and controlable.

Now in order to make sure the value can only be modified once per time it is set in the model, we add a variable to tell us when a particular readerset or writerset is used. This value is true only at the time a value is assigned to the set variable it belongs to. If the value is false we can therefore assume it

is not set so we have to keep the original value remembered as the "default" value.

SMV default read:

```
if(readused[i][j])
{
        getread[i][j] := setread[i][j];
        /*note that the defaultvalue does use next*/
        next(defaultread[i][j]) := getread[i][j];
}
else
{
        /*use the previous value since it has
          not been changed*/
        getread[i][j] := defaultread[i][j];
}
```

Adding the *readused* variable seems to be able to solve this problem but it actually moved the "SMV forgetting" to the *readused* variable. If it is not set to be true and not *explicitly* set to be false, then it can have both values. So we have to set it to false by default. It is not preferable to set it to false at every value of the current line variables for each read operation that could exist and for each memory type for obvious reasons. However we can not have contradictions in our model by stating it to be false at one place and true at another. In SMV this can easily be solved by using the *default* construct. It consists of two blocks of SMV code and allows you to assign a value to a variable in the first block in case it is not done in the second block. Therefore now it will look as follows also adding the modifying to the model:

SMV fault injection model:

```
if(readused[i][j])
{
        /*toss coin*/
        readattack[i][j] = {0, 1}
        if(readattack[i][j])
        {
                /*arbitrary value*/
                getread[i][j] := {MININT..MAXINT};
        }
        else
        {
                getread[i][j] := setread[i][j];
        }
        /*note that the defaultvalue does use next*/
        next(defaultread[i][j]) := getread[i][j];
}
else
{
        /*use the previous value since it has
          not been changed*/
        getread[i][j] := defaultread[i][j];
}
default
{
        readused[i][j] := 0;
}
in
{
        the Java specifically generated code
}
```

In this way no contradictions can occur and the model and the attacks can be modelled without having an unreadable model (for humans).

Also note that because there is no order in the SMV statements we can assign *readused* on a higher line number in the model's text then we have the statement which reads its value.

In order to make counting the attacks easier, either this is done in the model or in the property to be proved, we added a variable that would count the number of attacks that occurred for each type of attack. These counters should then be updated in a similar way to:

SMV error counting:

```
counterread : array {0..memtypes} of integer;
counterwrite : array {0..memtypes} of integer;

readattack[i][j] = {0, 1}
if(readattack[i][j])
{
        counter[i][j]++;/*something like this*/

        getread[i][j] := {MININT..MAXINT};
}
else
{
        getread[i][j] := setread[i][j];
}
```

The problem is that this can not be used because the update of the counter can not be placed inside a for loop. It would then assign a value to the variable multiple times in a single point in time. We solved this, using that the values indicating true equal 1, by the following code we could generate:

SMV error counting:

```
counterread : array {0..memtypes} of integer;
update loop
        ...
        readattack[i][j] = {0, 1}
        if(readattack[i][j])
        {
                ...
        }
        else
        {
                ...
        }
        ...
next(counterread[0]) := counterread[0]+readerattack[0][0]
        + readerattack[0][1] + readerattack[1][0] + ... ;
        /*adding all explicitly
        where 0 values are of no effect and therefore
        we add 1 for each "attack"*/
```

We can now add the possibility of resetting the device. For this we use a new variable and put all Java specifically generated code into one big if

statement. This saves a lot of trouble you would have using the default construct for this, since it is already used for the readused and writeused variables.

3.3.2 Framework

As might have become clear from all of the previous, there are multiple "levels" in the final generated model.

There is the part that is generated from the Java code, modeling the methods et cetera, with the addition of the reading and writing of variables at places in it. Then there is the "global" piece of the model where all the actual attacks are modeled and the registration coming with it is taking place. Also in the "global" section there is the initialization of the variables used and the definitions of some constants such as the *MAXINT* and *ARRAYSIZE* value, as discussed in previous sections. We call the "global" layer's layout the *framework* of the generated model.

Now the thing we left out in the story is how we start and stop the simulation of the Java methods, which are used and how the "local" part of the model is connected to the "global" part. How does it all fit in the framework? That is what we will discuss now.

In section 3.2.4 we explained about the way methods call each other and return to their callee. We used the same tricks to connect the parts of the model.

We created a new state of the system called *rest* that the system is in when none of the modeled methods are active. Now we decided to model the Java in a way that all public methods are to be verified. Therefore from the rest state we made sure each time all of the public methods could be called, after which the public method would return to the rest state again[4]. The SMV code for the rest state looks like the following:

Rest:

```
if(active=rest)
{
        next(active):={pubMethod1,pubMethod2,pubMethod3};
}
```

If public methods can also call each other of course the *returnTo* variables of the called method has to be set to "rest". In this way the model allows

[4]The returning is in itself an interesting property both for the correctness of the generated model and the "halting" of the system at some point (the absence of non terminating loops). This property is discussed in section 3.5

for a call to any public method at each time the system is in rest state. Also in this way, when you want to verify some property, you can state exactly, by the use of either public or private, which methods are to be verified. Of course this can also be done in any other way, but this is convenient and sufficient for the examples we considered. Also it reflects the idea of public methods.

Parameters to public methods

For the examples we want to model, it is sufficient to have all public methods take no parameters. If we do want to use parameters we change the method to private and created a new public method to call it with the correct parameters. This works fine but it is not the most sophisticated way to handle this of course. We do not need to go into finding practical solutions to this, but we can imagine there are some better solutions than the one we tested.

One particular thorough method you could apply is to call the method with any of the values of the parameters within its range[5].

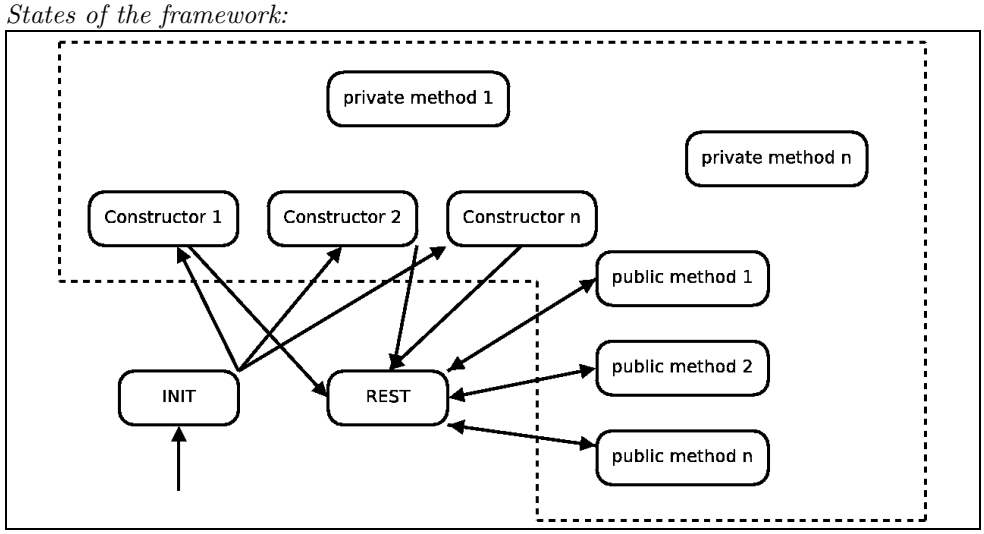
3.3.3 Constructor

Of course the rest state can only model the Java correctly *after* any of the constructors has been executed. How do we do that in the framework? We can do this in about the same way as we call the public methods, only we don't return to the *initial* state, but to the rest state, from which we don't return.

To make this more clear the diagram of the framework's states is illustrated in the following diagram:

[5]SMV will not allow for variables of undefined maximum size, so it is not the case that all string values have to be verified, which are theoretically infinitely many

States of the framework:



In this diagram all methods in the dotted area might call each other, apart from recursion, which is not allowed as discussed earlier. The system starts in the init state, where the variables are initialized, and then, via any of the constructors, goes and never returns from the rest state.

3.4 The SMV Generator

The experimental tool we wrote, to verify that the concepts as stated above are to some extend sufficient to automate the model checking for fault injection weaknesses, will be described in this section.

The very basis of the SMV generator is the code of the Jabstract project. It is written in Java (some version before Java 1.5) and has no documentation. The first thing we did was trying to get a grip on the Java code, which mainly consists of 62 classes, most of them representing nodes in the syntax tree.[6] Furthermore Jabstract contains some tools such as the *JavaParser* class, which has a method *CompilationUnit*. The CompilationUnit contains the information retrieved from parsing Java code, including the syntax tree.

We then created a class that generates the SMV by calling the CompilationUnit method of the JavaParser first and then using the resulting CompilationUnit instance to construct a new SMVModel class. This class is the outside layer of the generation process. It handles the package list, classes (inner classes could exist), and all other elements on a global level in the compilation unit. Then we created a class to parse methods which can

[6] More on how we figured it out in section 3.6.1

be used in the SMVModel class to generate the actual SMV code.

We parsed Constructors as special cases of a method, even though in the original syntax tree they are not of a similar usefull type. We must note here that we would not have created the abstract syntax tree as it is by Jabstract. Quite some things do not seem to make much sense even though the result is usable. This lack of equality of the constructors and methods is one.

When parsing a method we pass a Visibles class, with the appropriate root, as parameter. In this way all the handling of names can be done correctly.

Since we did not need to write a perfect SMV generator for all of Java 1.1, we decided to take the following approach to write the generator. After this global setup we wrote a method to parse any Statement. It calls the appropriate underlying method to actually generate the Statement's SMV. We then added the throw of an Exception in the case the Statement was not implemented yet:

statement:

```
private void generateSMV(Visibles visibles,
        Statement statement)
        throws SMVGeneratorException
{
        newBlock();//To start with a new SMVBlock
        MemoryModel.reset();//reset unique tickets
        if(statement instanceof Block)
        {
                generateSMV(visibles, (Block)statement);
        }
        else if(statement instanceof IfThenStatement)
        {
                generateSMV(visibles,
                        (IfThenStatement)statement);
        }
        else if(statement instanceof ReturnStatement)
        {
                generateSMV(visibles,
                        (ReturnStatement)statement);
        }

        ...

        else if(statement instanceof ForStatement)
        {
                generateSMV(visibles,
                        (ForStatement)statement);
        }
        else
                throw new SMVGeneratorException(
                        "statement "
                + statement.getClass().getSimpleName()
                + " is not implemented"
                + " to generate SMV yet");
}
```

In this way it was easy to know exactly which statements to implement first for my simple testing Java. We did the same thing for Expressions. All of this allowed me to recursively generate all the SMV required for the Java examples.

In order to add the modelling of the attacks easily we created two methods:

Reader model:

```
private ExpressionResult wrapReader (ExpressionResult expressionResult)
{
        if(expressionResult.isVariable()&&  //could be constant for example
                MemoryModel.isModelledRead( //attacks allowed on it
                        expressionResult.getVariable().getName()))
        {
                int memKind = expressionResult.getVariable().getMemoryType();
                //now get ticket
                int readerNr = MemoryModel.getNextReadIndex(memKind);
                //and finally set up the SMV and return the getread variable
                currentBlock().add(
                        new SMVLine("readerused["+memKind+"]["
                        +readerNr+"]:=1;"));
                currentBlock().add(
                        new SMVLine("readerset["+memKind+"]["
                        +readerNr+"]:="+expressionResult+";"));
                return new ExpressionResult("readerget["+memKind+"]["
                        +readerNr+"]");
        }
        return expressionResult;
}
```

Writer model:

```
private ExpressionResult wrapWriterReader (
        ExpressionResult expressionResultToWrite,
        ExpressionResult expressionResultToRead) throws SMVGeneratorException
{
        ExpressionResult read = wrapReader(expressionResultToRead);
        if(expressionResultToWrite.isVariable()&&
                MemoryModel.isModelledWrite(
                        expressionResultToWrite.getVariable().getName()))
        {
                int memKind =
                        expressionResultToWrite.getVariable().getMemoryType();
                int writerNr = MemoryModel.getNextWriteIndex(memKind);
                currentBlock().add(new SMVLine("writerused["+memKind+"]["
                        +writerNr+"]:=1;"));
                currentBlock().add(new SMVLine("writerset["+memKind+"]["
                        +writerNr+"]:="+read+";"));
                return new ExpressionResult("next("+expressionResultToWrite+
                        "):=writerget["+memKind+"]["+writerNr+"]");
        }
        else if (expressionResultToWrite.isVariable())
        {
                return new ExpressionResult("next("
                        +expressionResultToWrite+"):="
                        +wrapReader(expressionResultToRead));
        }
        throw new SMVGeneratorException("can not write to non-variable");
}
```

In the Java code we added come comments on the interesting lines to link them to previous sections.

We will now give an example of a method that handles the postfix expressions (like $a{+}{+}$).

PostfixExpression example:

```
private ExpressionResult generateSMV(Visibles visibles, PostfixExpression e)
        throws SMVGeneratorException
{
        currentBlock().add(SMVModel.commentsLine(e.comments));
        ExpressionResult it = generateSMV(visibles, e.it);
        String ret = "";
        if(e.op.equals("++"))
        {
                newBlock();
                it = new ExpressionResult(wrapWriterReader(it, it)+"+1");
                newBlock();
                ret = it.toString();
        }
        else if(e.op.equals("--"))
        {
                newBlock();
                it = new ExpressionResult(wrapWriterReader(it, it)+"-1");
                newBlock();
                ret = it.toString();
        }
        else
                throw new SMVGeneratorException(
                        "postfix expression operator "+e.op
                        +" has not yet been implemented");
        return new ExpressionResult(ret);
}
```

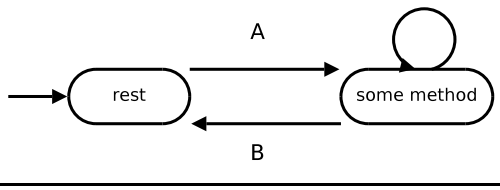
3.5 Verifying Properties

After the generation of the model we are not done. We need to state properties for the model checker to verify. As described in section 2.5.4 we can use "linear time" temporal logic [McM93] to state assertions on the model.

Obviously it depends on the property you want to verify how exactly to state it, but here we will provide some examples.

While testing the generator one of the properties that was interesting to verify is that *after "calling" methods from the rest state, you will always, at some point in time, want to return to the rest state again.* This can be displayed with the following automata:

Always return to rest:



The idea of this property is that taking transition A means taking transition B after a while.

In order to state this property we can use the *active* variable of the model. Using it we can state:

Always return to rest:

```
returnToRest :
assert G (
    (
        (!(active=rest))
        &
        G(writercounter[0]=0 & readercounter[0]=0)
    )
    ->
    (G F(active=rest))
);
```

Of course usually we have to include something about the number of reading and writing errors that are allowed to happen. A regular for loop will never terminate if the counter is never updated due to an infinite number of reading or writing errors. That is why the writercounter and readercounter are used here.

Another way of stating this property, and for the model checker a more easy property to verify, is the following:

Always return to rest:

```
returnToRest :
assert G (
    (
        (!(active=rest))
        &
        G(
          !readerattack[rx][ry]
          & !writerattack[wx][wy]
         )
    )
    ->
    (G F(active=rest))
);
```

where rx, ry, wx and wy should have to correct range to cover all "attack" values in the arrays. This indicates that never an attack takes place, thus leaving readercounter and writercounter to zero.

In the next section we consider the secure PIN as an example.

3.5.1 Secure PIN

To state that it is not possible to break the PIN security Java example is not that trivial. One way to do this is to implement a method in Java (call it

"checkPin"), where in a separate variable the number of consecutive incorrect tries is being recorded. This variable should not directly be influenced by read or write errors modelled of course. Suppose the implementation of the PIN example looks like this:

Defensive implementation:

```
...
if(get()>0)/*card not blocked*/
{
        decrease triesCounter;
        if(checkPin(pin))
        {
                set triesCounter to maxPinTries;
                ...
        }
        ...
}
...
```

then the checkPin might look like this

Recording actual tries:

```
private boolean checkPin(byte pin)
{
        if (pin!=correctPin) /*SAFE*/
        {
                attempts++;  /*SAFE*/
                return pin==correctPin; /*NOT SAFE*/
        }
        else
        {
                attempts=0;  /*SAFE*/
                return pin==correctPin; /*NOT SAFE*/
        }
}
```

where SAFE indicates that at that statement no attacks may happen and NOT SAFE means attacks might be allowed. In this way it is easy to really keep track of the actual tests executed on the PIN, which might be monitored from outside the chip.

The property stating the safeness of the card then can be stated like this for example:

Secure:

```
secure : assert G (attempts < maxPinTries);
```

Suppose *the three backup copies of the counter* can be attacked and the device will reset after an attack, the security property as stated above can be used as is demonstrated in chapter 4.

A simple Java class is shown with the generated model in appendix A to give a full example of a generated model using the SMV generator we wrote.

3.6 Problems faced

3.6.1 Syntax tree

When we started using the Jabstract project we had no clue on where to start due to the absence of documentation. There is a remark on the website (`http://www.doc.gold.ac.uk/~mas01sd/jabstract/`):

> Try something like java JavaTools.PrettyPrintWithComments test.java 0 where test.java is a Java program that you want to pretty print. You will end up with an extra file called TEMP which is the file without any comments.

So we looked at the code starting there. We figured out how to use the JavaParser to retrieve the abstract syntax tree of a Java file. Then we needed to find out what this tree actually looks like. This became a problem since there was actually not any clear tree structure. The nodes in the tree were not of some common type such as "TreeElement" by which you can recursively go through the tree. The first thing we did was creating such an interface for the abstract tree.

TreeElement:

```
public interface TreeElement {
        public boolean isLeaf();
        public List<TreeElement> getBranches();
}
```

Using this we can now print the abstract syntax tree of the Java file we used as input. This helped a lot in understanding the way Jabstract worked. That code looks similar to this:

TreeElement printing:

```
public static void printTree(OutputStream out,
        TreeElement el, String pre, String step)
                throws IOException
{
        List <TreeElement> branches = el.getBranches();
        out.write((pre+el.getClass().getSimpleName()
                +"\n").getBytes());

        if(branches!=null)
                for(;!branches.isnil();
                        branches = branches.tail()
                        )
                        printTree(
                                out, branches.head(),
                                pre+step, step);

}
```

3.6.2 Lists

The syntax tree elements contained Lists of Objects throughout and a lot of the fields of the classes had unclear names, at least to us. An example is a snippet of the OpListExpression as below:

Original OpListExpression:

```
public class OpListExpression extends Expression
{
        public String Operator;
        public List Expressions;

        public OpListExpression(List comments,
String o,List l)
        {
                super(comments);
                Operator=o;Expressions=l;
        }
}
```

To us there were quite some things unclear about the syntax tree and we decided we had to know more about it before actually using it to generate SMV. At the time we had not decided to use it for the rest of the research yet. We added comments to a lot of unclear variable names (or renamed

them). Also we rewrote some of the List class and the files coming with it to make use of the Java Generics of Java 1.5.
(`http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html`)

In short it allows you to specify what types are contained by a certain instance of a class that *can* contain all kinds of Objects. It prevents the use of casts everywhere in your code and makes it clear what types you can expect from a list or other "container". We added generics to all of the Lists.

We found that there was one place in the tree where it was not clear what types should exist in the List:

CompoundExpression:

```
public class CompoundExpression extends Expression
{
        public Expression e;
        public List <Object>AddedBits;
        ...
}
```

In the CompoundExpression AddedBits could contain both AddedBits and Expressions, which are not of some common type other than Object. Perhaps this indicates a mistake or bug? We did not look into it further since for this research it is irrelevant.

A lot more details of the syntax tree did not seem to be implemented to our taste, but for this research it seemed to work sufficiently well so we decided to use it for the generator.

The modification of List forced us to go through all the code where Lists are used and adapt all of it to use generics. This made us look at all the pieces of code that create syntax tree elements and gave us some idea of the workings of the Java parser. Because of the addition of generics, a lot of the unclear variables of syntax tree elements now became more clear, because the variable could only contain a certain type. Of course this was the case before, but now it stated so explicitly in the code making it much more clear.

3.6.3 Visibilities

The existence of block structures in Java, where variables only exist during parts of the execution, made us realize the use of implementing this properly. That is why we decided to use the Visibles class to keep track of variables and assign globally usable names for it.

Visibles snippet:

```
public class Visibles {
        private Vector<Visible> objects = new Vector<Visible>();
        private Visibles father = null;
        private String name = "";
        private String separator = "_";
        private int childCounter = 0;//used to generate unique names

        public Visibles()

        public Visibles(Visibles f)
        {
                father = f;
                name = father.getUniqueName();
        }

        public Visibles(Visibles f, String name)

        public String createName(String name)
        {
                return this.name+separator+name;
        }

        //returns unique name and updates childcounter
        private synchronized String getUniqueName() {
                return name+separator+childCounter++;
        }
}
```

3.6.4 Expression result

It helped in the generation to have all methods that generate SMV for some form of Expression to return a special class we called Expression result. Basically it allows you to return (a combination of)

- a String representation of something (an expression such as a fixed value or the result of an operation such as “a+b”)
- a variable
- a method

Furthermore the result can have a meaning in SMV or it can mean nothing at all in SMV. “a=1” would mean something when translated into SMV while “a+1” by itself would not.

Depending on which of the above the value of the expression is, different actions are allowed. For example “variable++” is allowed while “method++” is not. “method()++” is, however, because “method()” refers to some variable returned by the method instead of the method itself.

By allowing all three of these to be returned from the SMV generating methods, the methods that need to handle the result have the information they need.

The SMV representing only a variable could be an empty string because a single variable is not a valid SMV statement. When you have the expression "a=b+c", the result of it (b+c) has no meaning by itself, but in the expression "d=(a=b+c)" it has. This could be translated as "a=b+c; d=a+c;". So therefore a combination of the above kinds is also possible. We implemented it that way and it worked out to be a pretty clear solution.

3.6.5 State space

The attacks we wanted to model can be modeled in the way we described in section 2.2 and, yes, the model is still quite readable for humans. However the complexity of the model increased quite a bit.

If you generate a model in which all variables can, at any point, be attacked by introducing reading or writing errors, the model will get quite complex for larger Java code because of all the paths that can be taken in the model. Now what you can still verify also depends on the computer you use (to some extend) and how you tune the SMV model checker.

On the computer[7] we used we noticed that at some point the operating system started to swap memory to the harddisk, therefore slowing the calculations down a lot. In fact, we figured it would take too long to wait for. We then looked at the way the SMV model checker can be configured and we noticed the optimization option "BDD variable order sifting" which attempts to find an improved variable ordering. More on variable ordering can be found in [FAA01] for example.

This took some more calculations, therefore seemingly slowing it down first, but the improved variable ordering resulted in much less memory usage. Therefore much of the verification could be done after all on our computer by simply turning on the sifting.

We also added the functionality to configure the SMV generator to only allow for some fixed variables to be under attack, or to allow all but some to be attacked. Apart from the use of it for modelling the variables realistically under attack, this obviously saves a lot of state space, but one has to be careful in interpreting the results, since it might be the case that a certain (non verified) combination of variables still provides an attack.

3.6.6 Small specific problems

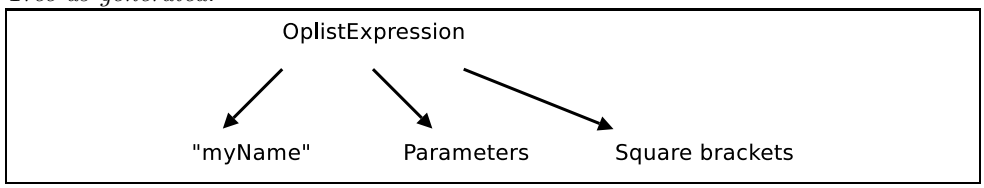
In this section we will go over some of the small and specific problems we faced during this research.

[7] More on the tools and material in the section 3.7

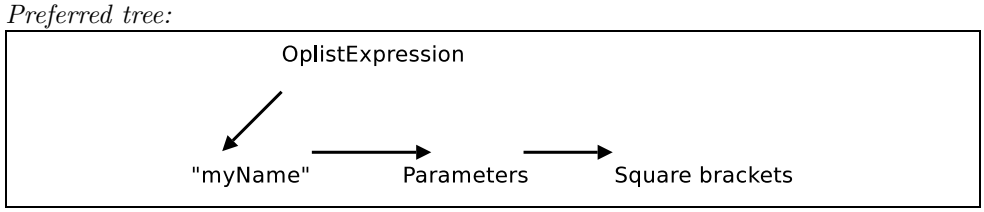
OpListExpression

One of the inconvenient things of the parse tree, as generated by the Jabstract tool, was that the OpListExpression contained expressions next to each other (that we would have rather seen in a hierarchical fashion).

Tree as generated:



Preferred tree:



Therefore the parent node had to handle the child instead of having the child handle itself.

Array handling

Concerning arrays we find two problems:

- a Java array's size might be unknown or variable
- assignments

The first is a problem because in SMV you have to specify the length of the array in advance. In fact, all variables have to have fixed sizes and be known at all times in SMV since it uses this in the verification process [McM01]. This contrasts to Java where, due to stack and heap kind of datastructures, it is possible to have a variable number of variables of variable size. (It is possible in Java to create, for example, n lists of n elements when input n is given) It is possible to create a stack like structure within a fixed sized list but this would cause problems in deciding the size of that, the amount of state space generated by the list and that the state space is not required since most of it will not be used all the time.

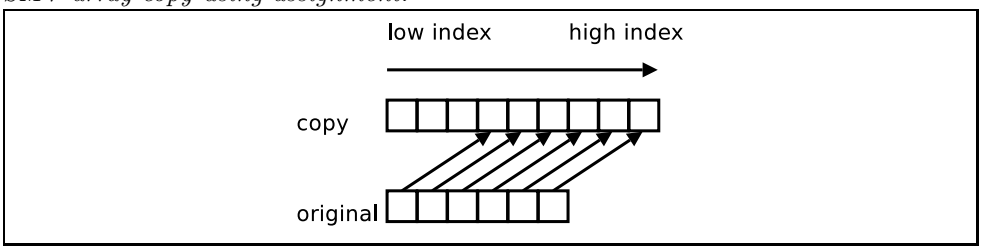
We solved the unknown size issue by using a fixed size definition that would allow for easy adaptation. All arrays have size "ARRAYSIZE" which

is defined in the first section of the SMV model to be of some value. If a person wishes to change this fixed size for some array(s), it is easy to find definitions of these in the model to adapt the size of single arrays and even easier to modify the size of all arrays at once. Obviously a generator can use these settings as parameter to the generation process. In this way the manual labour is not necessary.

The second problem is the copying. There are more ways to copy arrays in SMV. The most obvious way would be using the assignment :=; This however does not have to copy the array starting at the element with the lowest index [McM93].

Therefore, if arrays of different size were assigned, the elements could "shift position". The assignment of a short array to a long array might cause the following:

SMV array copy using assignment:



If the arrays are of equal size it is not a problem of course.

There are some solutions to this. The easiest and the one we implemented, because it solved the problem and the more complicated solutions did not seem much better for the purpose of this research, was to have arrays all of the same length. It is however not a problem to use the SMV *for* loop to copy the array from the lower index up, avoiding the difference in length problem.

Copying using a for-loop:

```
for(i = 0; i < maxIndexShortArray; i = i + 1){
        next(duplicate[i]) := original[i];
}
```

This would work but would not make the model more clear. It depends on the exact implementation of the model checker which of the solutions would require more state space. We tested it with the SMV model checker and it did not seem to make any difference.

The copying itself is solved using the same sizes for arrays, or the for-loop copying, but the referencing is not. This means that modifying the element

$a[1]$ after the assignment of $a = b$, $b[0]$ will also be changed in Java. This is a translation problem since SMV will copy the *content* of arrays therefore leaving $b[0]$ unchanged.

There seem to be more ways to do this but since it was not relevant for this research we did not look into it further. Some possibilities might include using references in SMV in much the same way as arrays are handled in machine languages, another is to duplicate generated assignments to all arrays that have to be updated. The Java code than has to be analyzed by the SMV generator in order to assign the right arrays. Possibly this last solution is not usable for all code.

This same problem occurs when handling arrays as parameters of methods which is discussed in section 3.2.5 on page 43.

Fields

Fields are the "global" variables of a Java class. Modelling fields add one small problem. The fields are often initialized before the constructor is executed. Consider the following example:

Fields:

```
public class Test
{
        private int testVar = 0;
        public Test()
        {
        }
        public int get()
        {
                return testVar++;
        }
}
```

All of these initialized fields in Java must also be initialized in the SMV model. However we can not use the *init* function to do so, since init will *only* state the value at the initial point in time. This means that at, say, the second point in time, the variable can have any value in its range. This means we cannot use the init function. The next function alone will also not work since it will not allow the variable to be changed:

Fields:

```
a : boolean;
init(a) := 0;//will not work since next(a) is not defined
next(a) := 0;    //will not work since next(a) is always
                 //defined and changes are not allowed
```

One solution is to use the next operator conditionally. Thus at some (and not all) point in time *before* the constructor is "executed".

An option is to add another value to the *active* variable. Another is to add initialization code (for all fields that require it) to the first constructor SMV line. In combination with the use of the init function this will solve the problem without adding complexity to the model.

Another, possibly better[8], solution to the problem is the use of the *default* construct in SMV. The following example demonstrates the use of it for fields (or any other variables):

Default next:

```
a : boolean;
init(a) := 0;//will not work since next(a) is not defined
default
{
        next(a) := a;
}
in
{
        /*actual code modelled*/
}
```

In this way all variables' values have been set initially and maintain their value unless explicitly stated otherwise.

Saving constructor

Another thing to be carefull of is the constructor when it comes to modelled attacks. It is (at least in the examples we used) assumed that the constructor models the factory initialization of the card. The attacks will occur after the initialization so therefore also in the model this should be the case.

We implemented this using a boolean "attackAllowed" variable. It is false until the rest state is reached where it will become and remain true.

[8]Depending on the model checker this adds to the rules of the model or cuts down the state space.

3.7 Used tools

In this section we will give some information on the exact environment of the research.

3.7.1 SMV

The SMV version we used is the experimental release written by Ken McMillan at Cadence Berkeley Labs release 10-11-02p1

The SMV model checker we used can be downloaded from
http://www.kenmcmil.com/smv.html

3.7.2 Jabstract

The Jabstract project provides a full Abstract Syntax and Parser for Java. It is created by Sebastian Danicic and can be found on
https://jabstract.dev.java.net/.

In its current form it can handle Java 1.1 which is sufficient for Java card software. The version we used is the one we downloaded during this research. According to the website
(`http://www.doc.gold.ac.uk/~mas01sd/jabstract/`) it was last updated on the 29th of January 2005.

3.7.3 Java environments

The Java environment that Jabstract can handle is Java 1.1. This is sufficient for the current Java card software and generally basic Java code. We implemented the SMV generator in Java 1.5

3.7.4 PC and software

We did most of the work on Ubuntu Linux 2.6.20-15-386 on a 2500+ Mobile AMD Athlon XP-M with 512 MB Twinmos DDR-RAM, which, at this time, is not a supercomputer.

For the development of the SMV generator and the modifications of the Jabstract source code we used the Eclipse 3.2.2 SDK environment.

Chapter 4

Intuitive and defensive PIN implementation

4.1 Java code

The following is the entire Java code of the intuitive implementation extended with the methods checkPin, tryTrue and tryFalse to verify the safety property:

```
package pinattack;

import javacard.framework.JCSystem;
import javacard.framework.Util;

public class TryCounter {

        private  byte[] _temp;

        private  byte _max;

        private byte[] _c1;
        private byte[] _c2;
        private byte[] _c3;

        private byte maximumvalue = 3;

        private int wrongguesses = 0;

        public TryCounter() {
                _max = maximumvalue;
                setNA(_max);
        }

        public void arrayCopyNonAtomic(byte [] a, byte aos, byte[] b, byte bos, byte len)
        {
                for(int i = 0; i < len; i++)
                {
                        b[i+bos] = a[i+aos];
                }
        }

        private void setNA(byte bytevalue) {
                _temp[0] = bytevalue;
                arrayCopyNonAtomic(_temp, (short)0, _c1, (short)0, (short)1);
                arrayCopyNonAtomic(_temp, (short)0, _c2, (short)0, (short)1);
                arrayCopyNonAtomic(_temp, (short)0, _c3, (short)0, (short)1);
```

```
}

private void restore() {
        if(_c1[0] == _c2[0] && _c2[0] == _c3[0]) {
                return;
        }
        if(_c2[0] == _c3[0]) {
                _temp[0] = _c3[0];
                arrayCopyNonAtomic(_temp, (short)0, _c1, (short)0, (short)1);
                return;
        }
        if(_c1[0] == _c2[0]) {
                _temp[0] = _c1[0];
                arrayCopyNonAtomic(_temp, (short)0, _c3, (short)0, (short)1);
                return;
        }
        // All three different
        _temp[0] = _c3[0];
        arrayCopyNonAtomic(_temp, (short)0, _c2, (short)0, (short)1);
        arrayCopyNonAtomic(_temp, (short)0, _c1, (short)0, (short)1);
}

private byte get() {
        restore();
        if (_c1[0] == _c2[0] && _c2[0] == _c3[0])
                return _c1[0];
        return (byte)0;
}

private void decrease() {
        restore();
        if(_c1[0] == (byte)0) {
                return;
        }
        setNA((byte)(_c1[0]-1));
}

private void max() {
        restore();
        setNA(_max);
}

private boolean checkPinCode(byte pin)
{
        if(pin!=3)
        {
                if(wrongguesses < 7)
                        wrongguesses++;
        }
        else
        {
                if(wrongguesses < 3)
                        wrongguesses = 0;
        }
        return pin!=3;
}
private boolean tryPin(byte pin)
{
        if(get()!=0)//otherwise blocked
        {
                if(checkPinCode(pin))//check pin
                {
                        decrease();
                }
                else
                {
                        max();
                        return true;
                }
        }
        return false;
}

public void tryTrue()
{
        tryPin(3);
```

```
        }
        public void tryFalse()
        {
                tryPin(0);
        }
}
```

The Java code for the defensive implementation is only different in the following method:

Defensive:

```
private boolean tryPin(byte pin)
{
        if(get()!=0)//otherwise blocked
        {
                decrease();
                if(checkPinCode(pin))//check PIN
                {
                        max();
                        return true;
                }
        }
        return false;
}
```

And the final line of checkPinCode is changed to

Defensive:

```
return (pin==3);
```

Most of these methods are discussed in section 2.3. Some are added to verify the safety property of the intuitive and defensive implementation.

- The checkPinCode method is used to keep track of the actual number of times the PIN calculations take place using the _wrongguesses variable.
- The tryPin method is used to implement the intuitive or defensive method of checking the PIN
- The tryTrue and tryFalse methods are used to call the tryPin method with the correct PIN and an incorrect PIN respectively.

4.2 Verifying the implementations

We generated SMV models for the Java implementations as shown in section 4.1.

4.2.1 The secure property

We use the *_wrongguesses* variable to keep track of the actual number of times the incorrect PIN is verified consecutively. This variable must not be attacked, so we configure the SMV generator not to generate these attacks. This registration allows us to describe the safety property very clearly. The property we want to verify is

Secure in general:

```
secure : assert G (_wrongguesses < 4);
```

If this holds, the PIN will not be evaluated after three consecutive incorrect tries. This is exactly what prevents chances over $\frac{3}{10000}$.

The *secure* property should hold when all variables (apart from those used to update _wrongguesses) can be attacked by read and write attacks, both of which reset the device. These attacks then model the card tear attack.

4.2.2 Verification of secure

There are different choices when it comes to the variables under attack. We can configure the generator to allow only the three counter variables to be under attack. A stronger configuration allows the attack on any[1] of the variables. We tested both of these configurations.

We used the SMV generator to generate the appropriate model. We then tuned the model by hand in the following way:

- We set the ARRAYSIZE definition two be one instead of the default. Larger arrays are not considered in this model, so one will do.
- We changed the maximum tries definition from an *init* to a *next* definition. We did this because the experimental generator did not handle initial values of fields yet. This only occurs once. The init function can not be used because in SMV it will only define the value at the initial time, leaving it undefined for next times.
- We added the secure property, as stated above, to the model.

[1] Again, apart from the _wrongguesses variable

Defensive implementation

Using the preparations as stated above, we were now able to verify that, for the defensive implementation, indeed the secure property holds for the weakest configuration:

```
Model checking results
======================

secure ..................................... true

Resources used
==============

user time ..................................... 329.82 s
system time ..................................... 0.65 s
BDD nodes allocated ..................... 507800
data segment size ..................................... 0
```

It took about 329 seconds (approximately five minutes and thirty seconds) to verify this.

We were also able to verify the property using the stronger configuration.

```
Model checking results
======================

secure ..................................... true

Resources used
==============

user time ..................................... 544.9 s
system time ..................................... 0.62 s
BDD nodes allocated ..................... 1280509
data segment size ..................................... 0
```

The property took about 545 seconds to verify (approximately nine minutes) with this configuration.

Intuitive implementation

Apart from the defensive implementation, we also tested the intuitive example. This should result into a counter example. At the point(s) in time between the actual PIN verification and the decrease of the counter(s) the device can be reset, leaving the counter in the old state.

```
Model checking results
```

```
=========================
secure ..... false

Resources used
==============
user time ..... 31.51 s
system time ..... 0.06 s
BDD nodes allocated ..... 245192
data segment size ..... 0
```

The intuitive implementation, as expected, resulted in a counter example. This is a summary of the counterexample:

time	active	_wrongguesses
1	constructor	0
71	rest	0
72	tryFalse	0
74	tryPin	0
76	get	0
78	restore	0
86	get	0
96	tryPin	0
100	checkPinCode	0
107	checkPinCode	1
112	tryPin	1
116	decrease	1
118	restore	1
122	rest	1 here the device has been reset
123	tryFalse	1
125	tryPin	1
127	get	1
129	restore	1
137	get	1
147	tryPin	1
151	checkPinCode	1
158	checkPinCode	2
162	tryPin	2
167	decrease	2
169	restore	2
173	rest	2 here the device has been reset again
...	...	...
260	checkPin	4

After the constructor, where they are set to three, the counter variables are never changed. The device is reset every time while the device is executing the decrease method (the restore method is called from it). This is exactly the weakness described in section 1.1.

It took about thirty seconds to find this counter example.

4.2.3 Continue attacks on the defensive implementation

The running example we used, at least the defensive implementation, should have withstand at least the weakest configuration as described above, which it did. It also withstands the stronger configuration in which all variables can be attacked by any reset attack. Obviously it is interesting to also try other properties and other configurations.

The continue attacks are interesting for this example. It is not a card tear, as the device would then reset, but nevertheless it is interesting to see if the implementation can withstand it.

We configured the generator to allow only *write reset* attacks on the counters. Obviously if we allow three of them at any points in time, the secure property would not hold. If, for example in the decrease method, we write five to all three of the variables, we simply have five attempts to try the PIN.

We tested the following property:

Secure single write attack:

```
secureOnlyWriteAttack :
        assert G (
                (G((writercounter[0]<2)))
                ->(G(_wrongguesses < 4)));
```

This property states that (having a configuration that does not allow for read attacks) allowing for at most a single write continue attack on one of the counters, the implementation is still secure.

The outcome of SMV for this property is the following:

```
Model checking results
======================

secureOnlyWriteAttack ..... true

Resources used
==============

user time ..... 12032.9 s
system time ..... 16.44 s
BDD nodes allocated ..... 3454231
data segment size ..... 0
```

It took SMV much longer to verify this property (three hours and twenty minutes compared to nine minutes). We think this is caused by the amount of possibilities the continuation generates. A reset brings us back to the rest

state, from which the model only allows continuation to much of the states that already had to be considered.

Interestingly, this property holds even though the defensive implementation is not explicitly designed against these attacks.

Pushing the limits of what this defensive implementation can take we also tested the property

Secure two write attacks:

```
secureOnlyWriteAttack2 :
        assert G (
                (G((writercounter[0]<3)))
                ->(G(_wrongguesses < 4)));
```

Which is different from the previous because it allows for at most two attacks instead of one. This property does not hold:

```
Model checking results
======================

secureOnlyWriteAttack2 ..... false

Resources used
==============

user time ..... 17886.8 s
system time ..... 24.36 s
BDD nodes allocated ..... 4639411
data segment size ..... 0
```

It took about five hours to complete this verification, being more complicated than the previous one. The counter example can be summarized as follows:

method called	c1	c2	c3	wg	note
constructor	3	3	3	0	the variables are initialized to 3
tryFalse	2	2	2	1	decreased variables
tryFalse	1	1	1	2	decreased variables
tryFalse	0	(1)	(1)	3	using two write continue attacks
restore	1	1	1	3	tryFalse calls get calls restore
tryFalse	0	0	0	4	tryFalse completes the PIN check

The two values between brackets should have been zero, but these write operations are tricked into writing a one instead of a zero. Now the card should be blocked but it is not.

Calling tryFalse again, the get method calls the restore method which assumes the two values of "1" to be correct. It "repairs" the only correct value of $c1$. Now the three values are equal and the get method returns the value one instead of zero, indicating that the card is not blocked. The PIN is checked and we broke security. Instead of tricking the write operations to write a one, it could also be tricked into writing a higher value, thereby allowing many attempts on guessing the PIN.

The defensive implementation can withstand any reset attack or a single write continue attack. It will fail when two write continue attacks are permitted.

Looking at read attacks we used the following property, allowing at most one read continue attack:

Secure single read attack:

```
secureOnlyReadAttack : assert G (
        (G((readercounter[1]<2))) ->
        (G(_wrongguesses < 4))
        );
```

Since we allow such an attack only on the counter variables (memory type we defined to be number 1), we can use *readercounter[1]* and keep *readercounter[0]* out of the equation. This means just one time only one of the counter variables can be read incorrectly, while still allowing the card to continue execution. This property does not hold:

```
Model checking results
======================

secureOnlyReadAttack ..... false

Resources used
==============

user time ..... 30876.3 s
system time ..... 35.48 s
BDD nodes allocated ..... 6590441
data segment size ..... 0
```

Taking about eight and a half hours a counter example is found. The attack uses the decrease method:

The decrease method:

```
private void decrease() {
        restore();
        if(_c1[0] == (byte)0) {
                return;
        }
        setNA((byte)(_c1[0]-1));
}
```

The line "if(_c1[0] == (byte)0)" reads the first counter variable. This read is corrupted to yield the value zero, at the time the actual (uncorrupted) value is still three. The decrease method does not decrease the counters but returns instead. The counters all remain to have the value three, thereby creating a fourth attempt to find the PIN.

Chapter 5

Discussion and conclusion

5.1 Discussion

The goal of this research was to investigate the use of model checking for code like the example we used (as described in section 2.3). The SMV generator we wrote is not complete nor do we claim it to be fully correct. However, it is capable of generating a model representing the example Java code, including the representation of the environment that allows for attacks. Although not complete and possibly incorrect at some points, the generator shows that it is possible to automatically generate such a model which is still usable with the model checker in question.

The way we modelled the Java code and environment is possibly not the most ideal. The implementation of the SMV generator is not written to produce an optimal model. It is written to generate a model that could still be used to verify some properties. This "blunt" implementation already worked much better than we expected. Probably implementing a more sophisticated generator will be able to handle even more complex code and more complex configurations of allowed attacks.

We did, however, not generate the code *exactly* as it might be executed on the chip. The reason for this is that on the chip Java byte code is executed (see for example [TL99] for more information on Java byte code). Modelling the processor and the execution of the byte code would result in a very complex model with complicating elements such as stacks, buffers and program counters. Furthermore, if the model checker is able to find a counter example, it could be hard to trace the counter example back to execution of code segments and therefore it could be hard to find the problem in the code.

Apart from the complexity of the model, it also did not seem wise to create a model at this low a level because the implementation of the processor, or the Java virtual machine running on it, is not fixed. Different implementations of processors and Java virtual machines exist, perhaps with build in optimization and other properties differing from each other. The generation than has to be adapted for each version used.

Another advantage of a higher level model is the increased readability.

The Jabstract project seemed to work for this research, but we are not sure whether it is a decent foundation for a serious generator. For example the strange AddedBits of the CompoundExpression should serve as a warning (section 3.6.2).

Floating point and double primitive types might be hard to model. In Java cards they currently do not exist, so this is not a problem, but somewhere in the future or for other cards they might be important. Probably with tuning the generation process modeling them can be done, but we can imagine some serious problems with it when precision matters. In general large numbers will require more of the model checker, so too many of them will perhaps make it impossible to use the generated model.

One of the most remarkable things (to our opinion) about this research is that we did not really have to pay too much attention to the state space. From previous model checking research we know this is not usually the case. We think this indicates the serious potential of automatic model checking for problems like fault injection.

We have the feeling that some people hesitate to use model checkers because it is (relatively) new to them, even though the benefits might be clear. Automated model checking might help these people.

The use of generation is not limited to Java cards or even smart cards. Many other pieces of code can be tested with a generated model. Software for other chips and even small pieces of software for common personal computers might be tested.

It is not in all cases trivial to formally state exactly what it is you want to prove, but we think automatic generation of a usable model from code will be a strong aid in writing correct software. Especially if unreliable hardware (due to an attack or any other reason) can result in an environment that is hard to predict.

The properties to prove can be stated once and likely do not have to be changed very frequently afterwards (in contrast to the code under development). Therefore the properties, that are sometimes difficult to formulate, do not have to be completely rewritten all the time. Of course the model can, as the code, be manually changed, but this manual labour can easily re-

sult in all kinds of mistakes, where automatic generation can be consequent and, if properly implemented, consequently correct.

In the example we considered (chapter 4) we augmented the code with an extra variable and some methods to ease the specification of the property we want to verify. Because of the automatic generation such additions require hardly any effort. Testing robustness to different kinds of attacks is easy because of the automatic generation and the counter variables (such as readcounter, which keeps track of the number of read attacks that took place). In verifying the secure property for the running example we considered, the generation paid off. Creating models to verify all these properties and configurations by hand would have been a lot of work and sensitive to human errors.

5.1.1 Alternative approaches

We solved some problems thouroughly, such as the hierarchical Java name space compared to the flat name space of SMV. A more easy solution to this would be to simply not allow duplicate names in the Java source. Also complex expression could have been forbidden instead of handled. Indeed these solution would have costed less work, requiring more of the user (who then perhaps has to adapt the source manually).

We think that it simply does not hurt to implement these solutions in the generator and that these make the use of it easier. It has to be implemented only once. If we do not implement it, we will have to modify each Java source containing such variable names or expressions by hand.

Inline methods (replacing a call by the code of the method) could also have been used instead of implementing a calling mechanism. These could make the resulting model quite unreadable however and implementing this correctly requires some care as well.

The following example illustrates one of the complications using inline methods:

Inline difficulty:

```
int inline(int a, int b)
{
        a = b + 1;
        a = b + 10;
        return a;
}
void main()
{
        int c = 10;
        c = inline(c, c);
}
```

Obviously the result of the inline method should be 20 (10+10) and not 21 (10+1+10). Simply inserting the parameter in the code of the method will result in the incorrect value (21) as illustrated in the next example:

Inlined difficulty:

```
void main()
{
        int c = 10;
        c = c + 1;        /* a = b + 1;  */
        c = c + 10;       /* a = b + 10; */
        c = c;            /* return a;   */
}
```

The idea of SMVBlocks (multiple SMV statements that are "executed" at the same point in time) works but has some consequences. Especially when reset attacks are concerned. Implementing these does not hurt, but they should not always be applied.

The way attacks are now modelled (using the readerset, readerget, readersetDefault, readerused, readercheat and readercounter array) works but requires a lot of registration. All of these variables make the model quite complex. Possibly a more elegant solution that can be implemented in a new generator exists.

5.2 Conclusion

According to our findings, using model checking to verify robustness of code against attacks, as in the example we used, can be done and can, perhaps more importantly, be done by automatically generating a model from the code.

The SMV generator we wrote, as a proof of concept, is capable of producing an SMV model, representing the code and the "hostile" chip environment, that the Cadence SMV model checker can still verify the properties to prove of.

We think it could greatly help developers to automate model generation for such purposes and this research shows it is a realistic goal.

The entire process of model checking, including the creation of the model, can probably never be done fully automatically, but generating the model can aid to

- Speed up the process of model building, thereby saving time required for building correct code
- Prevent human errors in the model
- Save people from learning all about a model checker's details
- Lower the threshold for starting to use model checkers

This research does not show the full potential of automatic model generation but it shows that it can be done, at least for code such as the example from [HMP06] as discussed in chapter 4. We are able to generate a model for this code, allowing for the attacks it is designed to withstand. Of this model we are able to verify the security property in five to nine minutes, depending on the amount of variables we allow to be under attack.

Chapter 6

Future work

6.1 In general

It is possible to use model checking to verify the robustness of software against failure of or attacks on hardware. The SMV generator we wrote is merely a proof of concept and we do not guarantee its correctness. It is our opinion that a tool to verify the robustness can be implemented in a better way and can strongly support the implementation process of software that might have to deal with malfunctioning hardware or attacks.

6.2 Continuing this research

Referenced types are not at all times handled correctly by the generator we wrote. This can be done in a better way to fully support Java variables in the generation. Simply copying variables back and forth when passed as parameters is not even sufficient as becomes clear from this example:

Referenced variables:

```
public void assign(int [] a, int [] b)
{
        b[0] = 0;
        a[0] = 1;
}
public void main()
{
        int [] a = new int [1];
        assign(a, a);
}
```

Another thing to improve is that local variables are not used optimally. When not in use they can be reused thereby saving state space.

6.2.1 Objects

We have not really looked further at modelling Objects. This was not required for this research but it is interesting as a follow up study. SMV does support structures so probably it can be done. How it can be done efficiently is not trivial.

6.2.2 Dependencies within Java

One of the harder topics to analyse is the dependencies withing Java code. As we stated earlier in section 3.2.7 it sometimes is possible to "execute" some Java statements at the same time. These kind of techniques are already researched in different areas such as *parallel computing* (See for example [TB94] or [RC00])

The benefit could be enough to implement this, since in SMV the model would decrease in size quite a bit, depending on the Java code of course.

This is already a difficult subject, but what makes it even more complicated is that the Java statements (or parts of an expression for that matter) combined in a single block will also behave as a single block when a reset attack occurs.

Block under attack:

```
a = 0;
b = 0;
```

When these statements are modelled in the same SMVBlock, a reset attack on a will still allow for the assignment of zero to b. This is not the case if these statements are modelled as different blocks. The same holds for expressions such as the following:

Block under attack:

```
b = (a = 0);
```

6.2.3 Improvements on the generator

As it is, the generator does not allow for combinations of reset and continue attacks. Combining these might be interesting for some examples. Also the current version does not model read reset attacks exactly right. Now it

is possible to have the following situation in which the generated model is incorrect.

Read reset attack on field:

```
a = variableUnderReadResetAttack;
```

The variable a will now be updated with the incorrect value returned by the read operation. This assignment should not happen. This is not necessarily a problem. If the variable a is not used again until it is initialized again (as with most local variables) it is not a problem. However, when a is a field (global variable) which is not reset and used throughout, the behavior is modelled incorrectly, or at least not as we would expect it to be.

Appendices

Appendix A

Generation example

As a simple example of a generated model we use the following Java code:

```
public class Example
{
        private int n;
        public Example()
        {
                n=0;
        }
        public int get()
        {
                n++;
                return n;
        }
}
```

Using the experimental SMV generator the following SMV model is created:

```
/*generated SMV model from Example.java on 2007-06-21 17:11:40*/
#define MAXBYTE 8
#define MAXINT 8
#define ARRAYSIZE 2
#define LOCALVAR 0
#define FIELDVAR 1
#define NRCOUNTERTYPES 2
typedef COUNTERTYPES 0..(NRCOUNTERTYPES-1);
typedef INTEGER 0..(MAXINT-1);
typedef BYTE 0..(MAXBYTE-1);
module main()
{
        /*Example class (extends '')*/
        /*
        GLOBAL DECLARATION
        */
        _n : INTEGER;
        /*used to prevent fault during constructor*/
```

```
cheatingAllowed : boolean;
init(cheatingAllowed) := 0;
/*used to set the active method*/
active : {rest, get, constructor};
init(active) := constructor;
/*used to reset the device*/
reset : boolean;
init(reset) := 0;
/*the following SMV is used to manipulate the reading and writing operations*/
readerset : array COUNTERTYPES of array 0..0 of INTEGER;
readerget : array COUNTERTYPES of array 0..0 of INTEGER;
readersetDefault : array COUNTERTYPES of array 0..0 of INTEGER;
readerused : array COUNTERTYPES of array 0..0 of boolean;
readercheat : array COUNTERTYPES of array 0..0 of boolean;
readercounter : array COUNTERTYPES of INTEGER;
for(i = 0; i < NRCOUNTERTYPES; i = i + 1)
{
        init(readercounter[i]) :=0;
}
for(i = 0; i < NRCOUNTERTYPES; i = i + 1)
{
        for(j=0;j<1; j=j+1)
        {
                if(readerused[i][j]){
                        if(cheatingAllowed){
                                readercheat[i][j] := {0,1};
                        }
                        else
                        {
                                readercheat[i][j] := 0;
                        }
                        if(readercheat[i][j])
                        {
                                readerget[i][j] := {INTEGER};
                        }
                        else
                        {
                                readerget[i][j] := readerset[i][j];
                        }
                        next(readersetDefault[i][j]):=readerget[i][j];
                }
                else
                {
                        readerget[i][j]:=readersetDefault[i][j];
                        readercheat[i][j] := {0};
                }
        }
        if(readercounter[i]+readercheat[i][0]<MAXINT)
        {
                next(readercounter[i]) := readercounter[i]+readercheat[i][0];
        }
        else
        {
                next(readercounter[i]) := MAXINT-1;
        }
}
writerset : array COUNTERTYPES of array 0..0 of INTEGER;
writersetDefault : array COUNTERTYPES of array 0..0 of INTEGER;
writerget : array COUNTERTYPES of array 0..0 of INTEGER;
writerused : array COUNTERTYPES of array 0..0 of boolean;
writercheat : array COUNTERTYPES of array 0..0 of boolean;
writercounter : array COUNTERTYPES of INTEGER;
for(i = 0; i < NRCOUNTERTYPES; i = i + 1)
{
        init(writercounter[i]) :=0;
}
for(i = 0; i < NRCOUNTERTYPES; i = i + 1)
{
        for(j=0;j<1; j=j+1)
        {
                if(writerused[i][j]){
                        if(cheatingAllowed){
                                writercheat[i][j] := {0,1};
                        }
                        else
                        {
```

```
                                writercheat[i][j] := 0;
                        }
                        if(writercheat[i][j])
                        {
                                writerget[i][j] := {INTEGER};
                        }
                        else
                        {
                                writerget[i][j] := writerset[i][j];
                        }
                        next(writersetDefault[i][j]):=writerget[i][j];
                }
                else
                {
                        writerget[i][j]:=writersetDefault[i][j];
                        writercheat[i][j] := {0};
                }
        }
        if(writercounter[i]+writercheat[i][0]<MAXINT)
        {
                next(writercounter[i]) := writercounter[i]+writercheat[i][0];
        }
        else
        {
                next(writercounter[i]) := MAXINT-1;
        }
}
if(1&(0|readercheat[0][0]|writercheat[0][0]|readercheat[1][0]|writercheat[1][0]))
{
        next(reset):=1;
}
else
{
        next(reset):=0;
}
default
{
        for(i = 0; i < NRCOUNTERTYPES; i = i + 1)
        {
                readerused[i][{0..0}] := 0;
                writerused[i][{0..0}] := 0;
        }
}
in
{
        /*
        the methods are defined below:
        */
        /*
        METHOD get
        */
        /*current line number*/
        get_CL : 0..4;
        init(get_CL) := 0;
        /*return from get*/
        get_returnTo : {rest};
        /*return value*/
        get_value : INTEGER;
        /*
        METHOD constructor
        */
        /*current line number*/
        constructor_CL : 0..3;
        init(constructor_CL) := 0;
        /*return from constructor*/
        constructor_returnTo : {rest};
        init(constructor_returnTo) := rest;
        if(reset)
        {
                next(get_CL):=0;
                next(constructor_CL):=0;
                next(active):=rest;
        }
        else
        {
                if(active=rest)
```

```
                        {
                                next(cheatingAllowed):=1;
                                next(active):={get};
                        }
                        else if (active=get)
                        {
                                if(get_CL=0)
                                {
                                        next(get_CL):=1;
                                }
                                else if(get_CL=1)
                                {
                                        readerused[1][0]:=1;
                                        readerset[1][0]:=_n;
                                        writerused[1][0]:=1;
                                        writerset[1][0]:=readerget[1][0];
                                        next(_n):=writerget[1][0]+1;
                                        next(get_CL):=2;
                                }
                                else if(get_CL=2)
                                {
                                        next(get_CL):=3;
                                }
                                else if(get_CL=3)
                                {
                                        next(get_value):=_n;
                                        next(get_CL):=0;
                                        next(active):=get_returnTo;
                                }
                                else if(get_CL=4)
                                {
                                        next(get_CL):=0;
                                        next(active):=get_returnTo;
                                }
                        }
                        else if (active=constructor)
                        {
                                if(constructor_CL=0)
                                {
                                        next(cheatingAllowed):=0;
                                        next(constructor_CL):=1;
                                }
                                else if(constructor_CL=1)
                                {
                                        writerused[1][0]:=1;
                                        writerset[1][0]:=0;
                                        next(_n):=writerget[1][0];
                                        next(constructor_CL):=2;
                                }
                                else if(constructor_CL=2)
                                {
                                        next(constructor_CL):=3;
                                }
                                else if(constructor_CL=3)
                                {
                                        next(constructor_CL):=0;
                                        next(active):=constructor_returnTo;
                                }
                        }
                }
        }
        returnToRest : assert G ((G((readercounter[{0..(NRCOUNTERTYPES-1)}]=0)
                                &(writercounter[{0..(NRCOUNTERTYPES-1)}]=0)))->((!(active=rest))
                                -> (G F(active=rest))));
}
```

Of which the returnToRest property holds. A more interesting example is discussed in chapter 4 (without the actual generated model, due to its size).

Bibliography

[BBE+99] Michael Baentsch, Peter Buhler, Thomas Eirich, Frank Horing, and Marcus Oestreicher. Javacard-from hype to reality. *IEEE Concurrency*, 07(4):36–43, 1999.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[FAA01] Karem A. Sakallah Fadi A. Aloul, Igor L. Markov. MINCE: A static global variable-ordering for SAT and BDD. In *International Workshop on Logic & Synthesis*. University of Michigan, June 2001.

[HMP06] Engelbert Hubbers, Wojciech Mostowski, and Erik Poll. Tearing Java Cards. In *Proceedings, e-Smart 2006, Sophia-Antipolis, France, September 20–22*, 2006.

[JG05] Guy Steeke Gilad Bracha James Gosling, Bill Joy. *Java(TM) Language Specification, The (3rd Edition)*. Prentice Hall, 2005.

[JRBH92] K. L. McMillan D. L. Dill J. R. Burch, E. M. Clarke and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Information and Computation*, number 98, pages 142–170. Elsevier, June 1992.

[LAMD04] V. Prakash L. A Mohammed, Abdul Rahman Ramli and Mohamed B. Daud. Smart card technology: Past, present, and future. *IJCIM*, 12(1):12–22, 2004.

[McM93] K. L. McMillan. The SMV language. 1993.

[McM99] K. L. McMillan. Getting started with SMV. 1999.

[McM01] K. L. McMillan. The SMV system. 2001.

[RC00] Leo Dagum David Kohr Dror Maydan Jeff McDonald Rohit Chandra, Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[RNS+04] K. Rothbart, U. Neffe, Ch. Steger, R. Weiss, E. Rieger, and A. Muehlberger. High level fault injection for attack simulation in smart cards. *ats*, 00:118–121, 2004.

[TB94] S. Turner and A. Back. General purpose optimistic parallel computing, March 1994.

[TB02] Sriram K. Rajamani Thomas Ball. The SLAM project: Debugging system software via static analysis. pages 1–3, 2002.

[TL99] Frank Yellin Tim Lindholm. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall, 1999.

[Wit02] Marc Witteman. Advances in smartcard secutity. 2002.