Modelling Java Card Applications With Defensive Measures In UPPAAL

SW10 Project
Group DES108F15
Department of Computer Science
Aalborg University
June 2 2015

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
Telephone: +45 9940 9940
Telefax: +45 9940 9798
http://cs.aau.dk

Title:

Modelling Java Card Applications With Defensive Measures In UPPAAL

Theme:

Distributed Embedded Systems

Project period:
P10, Spring Term 2015

Project group:
DES108F15

Participants:
Anders Kaastrup Vinther
Jakob Jørgensen

Supervisors:
René Rydhof Hansen
Mads Christian Olesen

Circulation: 5

Page count: 73

Appendix count and type: 2, Model Parser, Instructions of the Operational Semantics

Synopsis:

Cosmic rays and their impact on circuits have inspired attackers to use bitflip attacks to make arbitrary jumps in program execution. Some of the defensive mechanisms currently available to Java Cards to avoid this kind of attack have inherent vulnerabilities, it is one such defensive strategy this report will focus on, because it is currently used within the industry. The report demonstrates how UPPAAL can be a strong ally for program analysis. To reduce the work-burden, programs are based on the operational semantics defined in [18]. This makes it necessary to create a toolchain consisting of a program generator that generates programs which employs this defensive strategy, a model parser that reads programs and outputs .xml models for use with UPPAAL.

Finished on June 2nd 2015

Preface

This report has been written by two students from the Department of Computer Science at Cassiopeia, Aalborg University for the 10th semester Master's Thesis project.

Working with a language based closely on Java bytecode, we decided to model Java Card applications, but more specifically what would happen to them if a single event upset (bitflip) affected the program counter. Specifically how this would affect the control flow, and if the security measures in the program would be able to catch the event during execution. This model represents a single event upset where an attacker that did either not have the know-how or equipment to create a specific path through the program, but would be capable of deciding when the single event upset should take place.

Having created a model it could be used to determine whether or not UPPAAL would be a suitable tool for developers to evaluate their programs.

Enclosed with this report is a CD containing some of the files used in the project, specifically the source .pdf files that we were able to download. For website sources see the links in the bibliography. The report as was printed and delivered at the study office is also part of this CD in .pdf format. Finally, the program code written during this semester is on the CD and most importantly, the compiled .exe files along with some examples that can be tried. To recompile the parser you will need the boost library installed with C++[5].

We would like to thank Rene Rydhof Hansen and Mads Christian Olesen equally for their extremely capable help as counselors throughout this semester.

Signatures:

Anders Vinther

Jakob Jørgensen

Contents

1 Introduction **11**
- 1.1 The Project in Context . . . 12
  - 1.1.1 Software Based Protection . . . 12
- 1.2 The Defensive Strategy . . . 12
  - 1.2.1 A Bank Example . . . 13
  - 1.2.2 Attacking The Application . . . 15
  - 1.2.3 Java Implementation . . . 16
- 1.3 Initiating Problem . . . 18
  - 1.3.1 Problem Definition . . . 19

2 Preliminary Knowledge **21**
- 2.1 Control Flow Graphs . . . 21
- 2.2 Bit Flip Attacks . . . 22
- 2.3 The Model-Checker: UPPAAL . . . 23
  - 2.3.1 UPPAAL: SMC . . . 23
- 2.4 Java Card Virtual Machine Architecture . . . 24

3 Modelling a Java Card Application Control Flow **27**
- 3.1 The Example Program . . . 27
  - 3.1.1 Control Flow In Uppaal . . . 28
  - 3.1.2 Control Flow In UPPAAL With Modeled Attacks . . . 28
  - 3.1.3 Control Flow In UPPAAL-SMC With Modeled Attacks . . . 29
- 3.2 Modelling Programs . . . 35
  - 3.2.1 Modelling Start and End of Execution . . . 35
  - 3.2.2 Modelling Bad Bitflips . . . 35
  - 3.2.3 The Timing of a Bitflip . . . 36
  - 3.2.4 The Number of Bitflip Attacks . . . 36
  - 3.2.5 Success Criteria of a Bitflip Attack . . . 36

4 Modelling Java Card Defense Mechanisms **37**
- 4.1 Protective Measure . . . 37
- 4.2 Optimization of methods using Instructions . . . 40

5 Program Analysis with UPPAAL **43**
5.1 Tool Description . . . 43
5.1.1 Program Generator . . . 43
5.1.2 Model Parser . . . 47
5.2 Queries and Evaluation . . . 51
5.2.1 Balancing . . . 51
5.2.2 Execution Flow . . . 51
5.2.3 Distribution . . . 52

6 Reflection **57**
6.1 Toolchain Evaluation . . . 57
6.1.1 Model Parser . . . 57
6.1.2 Program Generator . . . 58
6.2 Alternative UPPAAL Models . . . 59
6.2.1 TIGA and Stratego . . . 59
6.2.2 Preliminary Results . . . 60
6.3 Changes to the Model . . . 60
6.4 Conclusion . . . 61

Bibliography **64**

I Appendices **65**

A Model Parser **67**

B Instructions of the Operational Semantics **69**

Chapter 1

Introduction

For as long as there has been electric memory in computing, there has been soft errors. A soft error is an error caused by cosmic rays, that strikes an electronic device just right, and causes some sort of erroneous behavior as result. These rays, as the name suggests, originate from space. The effects of this kind of erroneous behavior is entirely unpredictable however. This means that there is a potential for dramatic negative consequences to the program executing on the victimized device. For example, soft errors might corrupt calculations within the program, they might also cause the program to simply skip instructions being executed on the program. Hard errors on the other hand are much more severe in the sense that they corrupt hardware permanently. For example, memory cells can be corrupted by a hard error, caused by a similar kind of cosmic ray. A program using this device's corrupted memory cell will produce strange results permanently[17].

Ultimately, these are entirely unpredictable and undesirable errors. However given that the rays comes from outer space and is part of what is known as background radiation, it might be appealing to assume that this kind of error is so unlikely, that it just might not occur on non-spaceship electronics. Looking at soft and hard errors exclusively, it is true that it is an unlikely, but not impossible phenomenon[17]. However there exists an entire field of attack based around this phenomenon, which means that it is something malicious end-users can exploit and take a more sophisticated approach to.

In fact, because circuits are inherently vulnerable to energy particles, there exists a number of different styles of bitflip attacks, as shown in [6, 14, 18]. It is this area of attack this report is concerned with. The following chapter will therefore give a further introduction to the problem environment as well as manifesting the end goal. Furthermore, the chapter will give an overview of where this report continues, from the previous [18].

1.1 The Project in Context

The project is concerned with the Java Card technology with a relation toward the aforementioned notion of soft and hard errors and bitflip attacks - otherwise known as Single Event Upsets. In this report the term bitflip attack simply refers to an error caused by a malicious end user by way of introducing hard or soft errors to the Java Card circuit through a laser or other energy-ray emitting device.

The Java Smart Card is a kind of smart card which employs the Java Virtual Machine on a much smaller scale - namely as a smart card. There are certain hardware restrictions on this different platform, which means that it is not the full Java Virtual Machine that exists within this space, rather it is a subset of those instructions as described in [18, 15]. Certain defensive strategies are employed on Java Smart Cards to capture Single Event Upsets. What interests us the most, are the software based initiatives to improve Virtual Machine integrity, as opposed to improving hardware with various sensors, or simply building the chips in materials that is more resistant to energy particles [14, 17, 11]. For more information about the Java Card technology see [18, 15].

1.1.1 Software Based Protection

In this report we will generalize software based protection and see it as falling into either of two categories. Either the virtual machine is expanded in such a way that it becomes more self-aware, or certain code practices are upheld within a program to ensure the program is secure. The first category means that the virtual machine must be expanded with extra bytecode instructions that can detect when malicious behavior occurs, or similar but 'secure' instructions are executed. Either way, the virtual machine becomes smarter, and is in charge of ensuring its own security properties. There are two ways to employ safety features in this category. Either the virtual machine knows what bytecode instructions to execute, secure ones or insecure ones, depending on its own execution context, or a programmer has added flags to the program, that helps the virtual machine differentiate between secure and insecure blocks of code[3].

In the second category, already existing language constructs are employed, to achieve much the same goal. The key difference here is that this category of defensive mechanisms does not require an expanded virtual machine. This means that the application is easily more portable between Java Cards, as opposed to requiring a very specific virtual machine model[1].

1.2 The Defensive Strategy

The defensive strategy that was chosen to be analyzed for this project, is a method of protection that is used within the industry. The main procedure to take into account from [1], is the flagging process. The entire solution proposed in [1] consists of developer-written flags, a pre-processer, control-flow

graph analysis and more. However, these aspects are of little relevance to this project, because we will only be concerned with what is actually visible within a program's execution flow and what should be modeled as part of a program's control-flow graph. Incidentally, this makes only the flagging process interesting to us.

Developers set flags during their program's execution, which defines to the virtual machine, what is the expected control flow. When execution passes through a flagged zone, a counter is incremented, which is then later compared to some static/hardcoded value - this can be considered a check point. A mismatch between this incremented value and the static value, will then define to the virtual machine what is legal and illegal behavior. Essentially program areas will have their own unique identifier, and a simple if-check comparing this unique identifier, with a static value will determine whether the control-flow has executed normally or some of it has been skipped. The strategy's goal is to capture changes in the control flow caused by a malicious end user.

1.2.1 A Bank Example

Consider for example a bank application that employs this defensive strategy. As an illustration of the example, see Figure 1.1. Squares signifies standard ATM Control Flow, with the ruby-shapes signifying the safety mechanisms from Section 1.2. Whether or not this is a realistic representation of the mechanisms in an ATM is irrelevant, the example should simply illustrate the general premise of this defensive strategy.

When program execution is about to reach a 'secure' block of code, the counter value is incremented to some value. In other words, a flag signifying a secure block of code is about to happen is set. This is shown in `Increment Counter`. Throughout the execution of this secure block, this value might be incremented multiple times, until finally a checkpoint is reached. After the first incrementation a `Validate Card` step is reached. This step is assumed to ensure that it is the correct kind of card that has been inserted into the ATM. After that, the `Increment Counter` state is reached and `counter` at this point holds the integer value 3. `Validate PIN` is assumed to take user input and validate whether it was the correct number, or not. Finally, the `Check Value` state is reached, here the counter is compared to a static value, and a branching will occur based on the boolean result. The `Security Exception` is thrown by the defensive mechanism, which will ultimately terminate applet execution.

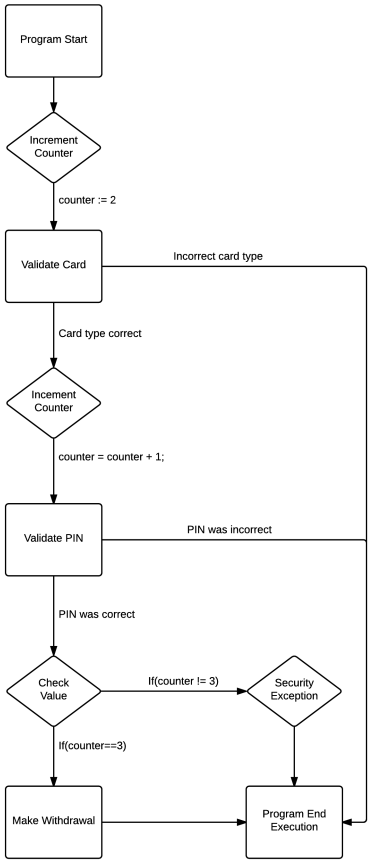


Figure 1.1: ATM Example.

1.2.2 Attacking The Application

There exists two kinds of control-flow in the system illustrated in Figure 1.1. The first control flow is the control flow that the bankers wants the user to have and is what is on display in the figure, but there is also another control flow that has not been shown. This control flow is what the hacker wants. As illustrated in Figure 1.2 the dotted lines show the transitions through the system that allows the hacker to withdraw cash, by escaping or evading entirely the security mechanisms written by the developers. This is known as a control flow attack. It is achievable by manipulating the PC-register on the smart card with some external force[10].

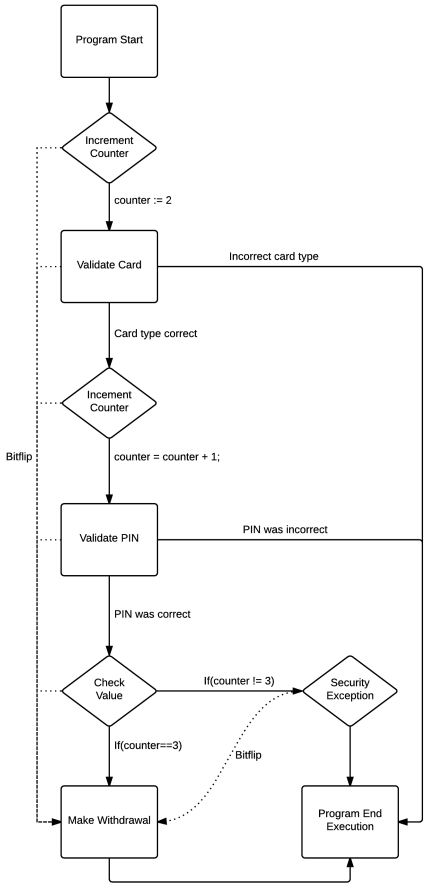


Figure 1.2: ATM Example with Bitflip Opportunities.

This is the kind of attack that the defensive strategy in Section 1.2 will try to stop. However, by adding this defense-specific code (incrementation, validation/checking and exceptions) the logical observation would be that the application has grown larger in terms of instruction space, and therefore there

is a larger area to target with this kind of control flow attack. The dynamics at play are similar to the combined attacks explained in [18, 12], where instead of program-lines as is the case in Section 1.2, the objects created simply take up more space in memory to make the target easier to hit.

1.2.3 Java Implementation

The following shows two examples in pseudo code of how the defensive strategy can be implemented in Java. The first example is the example that we have seen used in the industry Listing 1.1. The other example is shown in Listing 1.2, and is our adaptation of the model as will be modeled throughout this report. Notice that each class is equipped with a `counter` variable. This is the variable that is manipulated as illustrated in Figure 1.2. The importance in this example is when the `counter` is incremented. The counter is incremented in both paths true and false in an if-statement, and every check statement has the opportunity of throwing a security exception. If the counter does not match the specific value, a control flow attack has been committed.

Listing 1.1: Object oriented implementation in Java.

```
1  public class Example
2  {
3    public static int counter = 0;
4    /*
5      . More Variable Declarations
6    */
7
8    public static void main(String[] args)
9    {
10     counter = 0;
11     vc = validateCard();
12     checkCounter(1);
13     vp = validatePin(input);
14     if(vc && vp)
15     {
16       checkCounter(2)
17       makeWithdrawal();
18     }
19   }
20
21   // ATM example
22   private static void makeWithdrawal()
23   {
24     /*
25     . User privileges given, make withdrawal
26     */
27   }
28
```

```
29    private static boolean validateCard()
30    {
31      if(card is proper)
32      {
33        incrementCounter();
34        return true;
35      }
36      else
37      {
38        incrementCounter();
39        return false;
40      }
41    }
42
43    private static boolean validatePin(int input)
44    {
45      if(pin is correct)
46      {
47        incrementCounter();
48        return true;
49      }
50      else
51      {
52        incrementCounter();
53        return false;
54      }
55    }
56
57    // Counter specific code
58    private static void checkCounter(int value)
59    {
60      if(counter != value)
61      {
62        throw new security exception();
63      }
64    }
65
66    private static void incrementCounter()
67    {
68      counter++;
69    }
70
71    private static void incrementCounter(int value)
72    {
73      counter += value;
74    }
75  }
```

Listing 1.2: The implementation in our models.

```
1  public class Example
2  {
3    public static int counter = 0;
4    /*
5      . More Variable Declarations
6    */
7
8    public static void main(String[] args)
9    {
10     counter = 2;
11     if(card == "visa")
12     {
13       counter  ;
14       if(counter == 1)
15       {
16         if(pin == 4242)
17         {
18           counter  ;
19           if(counter == 0)
20           {
21             /*
22             . Withdraw money
23             */
24           }
25           else
26           {
27             throw security exception();
28           }
29         }
30       }
31       else
32       {
33         throw security exception();
34       }
35     }
36   }
37 }
```

Because we are only concerned with singular methods, the defensive mechanism will be modeled as in Listing 1.2. This is to avoid function calls, and keep the counter manipulation strictly as statements.

1.3 Initiating Problem

As described in [18] and in Section 1.1 there are different strategies to use when it comes to securing the integrity of a Java Card Applet. The strategy of interest is the method based on [1] and explained in Section 1.2, because it

potentially opens up other avenues of insecurity, that are interesting to explore. Ultimately, developers seeking to increase the degree of security in their applets, that employs this method, might wind up making their application insecure as result, illustrated in Figure 1.2.

1.3.1 Problem Definition

This report will provide a method for analyzing the security properties of Java Card applications that employs this specific defensive strategy. The proposed solution is UPPAAL SMC, which will be used in combination with a toolchain that consists of a model parser and Java Card program generator based on bytecode instructions from [18].

Chapter 2

Preliminary Knowledge

Because this report carries on from where [18] left off, this chapter will give an overview of the preliminary knowledge that is necessary to fully understand this report. The chapter should be seen as an analysis on a few but important topics required to understanding the solution proposed in this report.

2.1 Control Flow Graphs

Control Flow Graphs (CFG) are used in computer science to illustrate the execution flow of programs or methods. It is up to the modeller to determine what each individual vertex in such a graph represents. The following Figure 2.1 illustrates the control flow of a simple while-loop construct

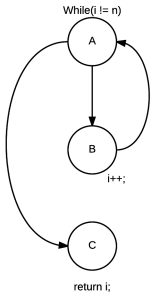


Figure 2.1: A While-loop Construct.

In this report a vertex corresponds to a bytecode instruction from the operational semantics defined in [18]. The program in Listing 2.1 has a corresponding control flow graph illustrated in Figure 2.2.

Listing 2.1: A Small Addition Program.

```
0 load 0
1 load 1
2 math+
3 return
```

The numbers to the left of Listing 2.1 corresponds to the assumed PC-register value of each instruction. This is a generalization because not all bytecode instructions are exactly 1 byte long - this is typically dependent on the number of operands (located on the operand stack), a bytecode instruction takes.

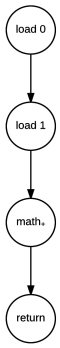


Figure 2.2: Addition Program.

2.2 Bit Flip Attacks

A bitflip attack is a type of single-event upset where a bit is forced to change its logical state from zero to one (or the other way around). A Multi-Event Upset refers to the same thing happening to several bits at the same time. Bitflips are possibly caused and are often associated with cosmic rays, variations in the clock signal and the supply voltage[10, 17, 11].

A variation of this has been proven viable against smart cards, writing on the EEPROM using probing needles[2]. The following is a definition of what is a control flow attack, and therefor the focus of this attack.

- Control flow attack: In the Java Card Environment the Program Counter is a register in charge of pointing to which line of code is being executed. The base idea is that alterations on the PC makes it possible for the attacker to make jumps in the control flow. This changes which program instruction is being executed. By doing this, it is possible to bypass security checkpoints in an application. On a model (control flow graph) this can be simulated by adding edges between vertices, as seen in the example at Section 3.1.2.

2.3 The Model-Checker: UPPAAL

UPPAAL at its core is a modelchecking tool based around timed automata. The variations of UPPAAL all make use of the same core, which includes the `.xml` storage system used to define and save automata, but also the timed core. This means that models in standard UPPAAL are also usable in other variations of UPPAAL, however not necessarily vice versa (in the case of UPPAAL SMC it works both ways).

Because UPPAAL models timed automata, the concept of clocks and time is important to all models. Time has to be modelled in some shape or form in any give model. Time is not relevant to the concept of this report's proposed solution however. For this reason, most states are flagged as `urgent` which simply means that time units are not allowed to pass while in a state with this flag. Essentially it means that UPPAAL has to take a transition out of the state as soon as the state has been entered.

UPPAAL has a strong verification engine based around a language known as timed computation tree logic (TCTL). The language that UPPAAL employs is simply described as a query language however, and it is this term that will be used throughout this report [4].

The verification engine calculates the boolean value of a query. This query language will allow a user to ask different questions about their model in the range of reachability based properties. For example; *Is it possible to reach a given state under certain conditions?* It is possible to have the property's truth value illustrated with a diagnostic trace. This illustrates the path for the user to further the understanding of the model as well as the query [4].

2.3.1 UPPAAL: SMC

SMC is a Statistical Model Checking tool based on UPPAAL. SMC is built into later versions of UPPAAL by default and is thus no longer an independent tool [9]. UPPAAL SMC supports statistical analysis on different queries. For example, it is possible to query a model based around UPPAAL SMC, what is the probability of some condition being satisfied. To achieve this, UPPAAL SMC is dependent on probability weights on different edges, the modeller should therefor have a reasonable understanding of what is the likelihood of a given edge to be chosen, otherwise the probabilities will not necessarily correspond to the real-world. UPPAAL SMC enables further flexibility in terms of UPPAAL's non-determinism. Like TIGA, UPPAAL SMC's query language has been expanded to capture these extra statistical opportunities [9]. The below highlights some of the key-features of UPPAAL SMC, for the full list of features, see: [9].

Confidence Intervals and Hypothesis Testing

One of the most notable additions to the query language with UPPAAL SMC in regards to this project, is the probability functionality. `Pr` is a keyword that

when used in the context of testing a property, tests the model in a necessary amount of runs, until a desired confidence level has been reached. The mathematics behind it is known as confidence interval testing, it is known simply as hypothesis testing in UPPAAL SMC. The output of running a `Pr`-based query on a property is a probabilistic interval, that some property will hold. UPPAAL SMC tries to get a confidence level of 95% on this probability interval.

Piecing this information together, the correct understanding of this interval is that 95% of the time, the true probability exists within this probabilistic interval. There will be cases where the interval is larger or smaller, which is why the intervals change for each run. However, the changes are rarely drastic enough to make the result meaningless in this context. Basically, though the interval will change, there is a 95% chance that this new interval will still contain the true probabilistic value.

Hypothesis testing is bounded in UPPAAL SMC, either bounded by time via clocks, or bounded by a set number discrete steps in the model. Executing such a query will allow an analyst to make statements about the probabilistic distribution in regards to this bounded value [9].

Diagrams and Simulations

Hypothesis testing is not the only addition to the UPPAAL SMC query language, there are other additions as well. One especially is of interest, namely `simulate`, which like hypothesis testing is bounded. Essentially it allows a user to test their model across a specific number of simulations. The bounding may either be discrete steps in the model, or time. Finally, simulate can test many different state-based properties in a model, each state property being comma separated.

In UPPAAL SMC every query can reveal its statistical distribution via diagrams, giving a user opportunities to further their understanding of their own models as well as their queries by having the results visualized.

2.4 Java Card Virtual Machine Architecture

There are two kinds of Java Cards as described in [18]. One of the versions is a version for the future, where there is a significant boost to the platform's hardware capabilities. This report however focuses on contemporary Java Cards, based on version 2.

The hardware specifications can be seen in Table 2.1. The capabilities of a Java Card based on its hardware is limited with little space on RAM (runtime data), ROM (operating system) and EEPROM (application packages).

Furthermore the virtual machine lacks certain program constructs, such as threading[16].

Figure 2.3 displays the runtime data area of the Java Card Virtual Machine (JCVM). PC is the Program Counter, one exists for every thread and it is a register containing the address of the instruction currently being executed

CPU:	8/16-bit
RAM:	2kb
ROM:	48-64kb
EEPROM:	8-32kb
I/O Interface:	Serial
Transfer Rate:	9.6-30kb/s
Duplex:	Full

Table 2.1: Traditional Smart Card Hardware[13].

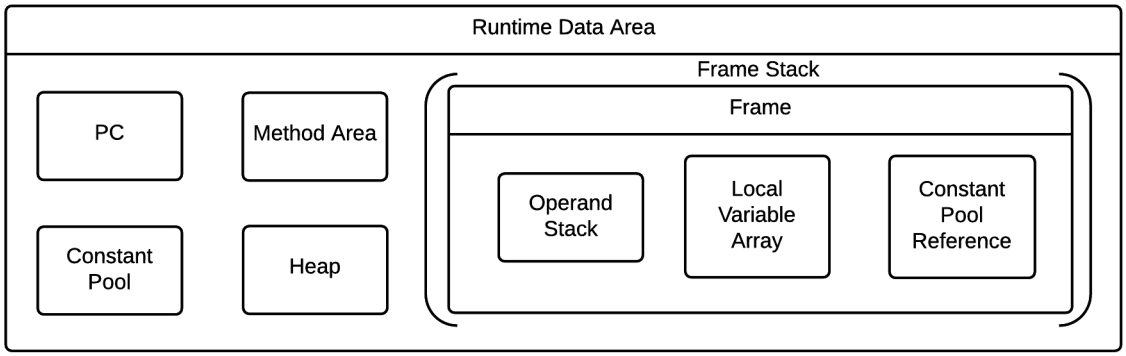


Figure 2.3: The Runtime Data Area of the JCVM, based on [18].

and a return address, as the JCVM on traditional java cards does not support multi-threading, only one PC exists.

The method area stores the code for methods and constructors, along with field and method data and the run time constant pool. The run-time representation of the constant pool is specific to the class currently being executed, it contains things such as numeric literals and field references.

The heap is where all class instances and arrays are allocated. The frame stack holds the frames of every method that has been called but has not finished its execution yet, with the most recent one being on top.

Each frame contains things necessary to a method, an operand stack, a local variable array, and a reference to the constant pool of the class that the method belongs to.

Operand stacks are used to temporarily store the values that instructions need during execution, while the local variable array store values for later use[16].

Chapter 3

Modelling a Java Card Application Control Flow

The following chapter will explain by way of a manual example, how a Java Card application's control flow can be modeled as a state machine. The example will be based on the operational semantics as formalized in [18]. This operational semantics is based on a JCVM-like language, so while there are some differences, they should prove negligible in regards to understanding the example. The operational semantics has the advantage of being formally defined in a way the bytecode instructions in the Java Card Virtual Machine specification is not.

3.1 The Example Program

Consider a small application whose purpose is to function like an ATM. Two pin codes are loaded and compared as integers, one of these is user input, whereas the other is a predefined integer on the card. In case the numbers are equal, the application would return one for true, otherwise the program returns zero, for false. Though not modelled in the below Listing 3.1, in case the comparison is true, the rest of the program should call some kind of log-in mechanism which gives the user privileges to make a monetary withdrawal. On the other hand, if the two numbers are not the same the app should terminate.

The flow of instructions is shown in Listing 3.1.

Listing 3.1: Example ATM Program.

```
0 load 1
1 load 2
2 cmp= 5
3 push 0
4 return
5 push 1
6 return
```

3.1.1 Control Flow In Uppaal

Figure 3.1 displays the control flow of the example in Listing 3.1, as modeled in UPPAAL. In this particular model each vertex represents an instruction in Listing 3.1, and the edges represent the normal control flow between the instructions. Tied together they display the possible execution flows of this method. The method starts in vertex zero and will finish in vertex 'end' where the method return has either returned 'one' from vertex six or returned 'zero' from vertex four. After this happens there is no further flow through the method. The example bears some resemblance to what was introduced in Figure 1.1. Since the method has an if-statement the model also has a split at vertex two, going to either vertices three or five. This is the comparison instruction from Listing 3.1.

Because UPPAAL is created to model timed automata, it is possible for execution to halt on any given vertex. To avoid this, all vertices are marked as `urgent` to ensure that a transition must be taken [4].

This models regular execution of a method. However, the model is not yet entirely complete because malicious execution flow has not yet been taken into account. Recall that malicious execution flow is when the model takes the possibility of control flow attacks into account, as described in Section 2.2 and shown in Figure 1.2. This is displayed in the following part of the example.

3.1.2 Control Flow In UPPAAL With Modeled Attacks

Figure 3.2 displays the control flow of example code, augmented with the possibility of bitflips to a PC. For simplicity the PC in this one example is only four bits. The process of modeling bitflips is done simply by remembering that the vertex name shows the line of code it represents. The PC points to these lines of code being executed. The easiest example to understand this, is flipping on PC zero. Zero is binary encoded as: (0 0 0 0). Moving from the first to the last bit, an attack here can result in a jump in executions from zero to one (0 0 0 1), two (0 0 1 0), four (0 1 0 0) and eight (1 0 0 0). Eight however, is an instruction that is not part of the current instruction space and must be handled in a special way, for now it is simply ignored. These possible jumps are marked on the model as edges between the vertexes. Note that these attacks are not possible from edges where execution of a method or program ends, like

return instructions. This is because attacks are modeled as happening after the instruction has been executed.

We can use this model to show where the code is vulnerable to this particular form of control flow attacks. Security measures are not of much value if they can be bypassed without execution after all.

Modelling the Attacker

The attacker is modeled in Figure 3.3, it is shown as a range of choices of where the bitflip is going to take place. The model of the method waits to start its execution until after this choice has been made. This is simulated by listening for a channel broadcast between the start vertex and the first instruction vertex.

Every edge from an instruction vertex is protected with guards to make sure that if an attack is supposed to happen there it cannot continue along normal control flow. If the attack is not to take place the bit flip edges cannot be taken because of this guard. These measures should ensure that it is equally likely for all vertices to be attacked. Updating the 'x' value at the time of the attack will assure that the attack only happens once, in case of loops in the model.

The example will now continue with an adaption to UPPAAL SMC.

3.1.3 Control Flow In UPPAAL-SMC With Modeled Attacks

The example model is expanded with UPPAAL branch-points, the model can be seen in Figure 3.4. Branchpoints are the small circles next to an instruction in the figure. These smaller circles should not be confused with normal instructions. Branchpoints are used by UPPAAL-SMC to add more complicated non-determinism to a model. It allows us to weight outgoing edges from a branchpoint, determining whether or not one edge is more likely to be chosen over others. Since we do not have any statistical data to determine the likeliness of one bit being flipped over the other, every edge is weighted '1' which gives each edge a 1/8 chance of being chosen ($n/8$ where n attacks can go to 'end' from different bitflips to the same vertex).

Modeling programs in this fashion allows us to model an attacker who is capable of deciding which instruction he wants to attack from, but unable to pinpoint exactly which bit he flips. Using the built-in verifier in UPPAAL we can determine the probability of an attacker being able to launch a successful attack.

Because the attacker does not have control over which bit he flips, there is the chance of him landing on a program counter that points outside the scope of the method (PC 'zero' through 'six' in the example). Landing outside the scope of the method should result in the program crashing and thus the edges that do this points to the `BadFlip` vertex in the model. For now the chance of this happening is the only thing that prevents an attacker from successfully subverting the control flow. This also means that there is a chance of successful

subversion, to compare against the same model with the countermeasures described in Section 1.2. The example will be expanded in Chapter 4 with this defensive mechanism.

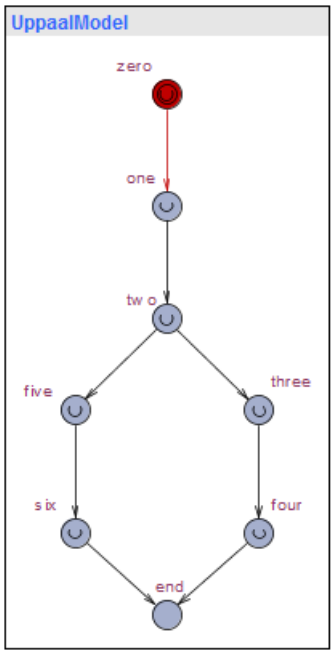


Figure 3.1: ATM example modeled in UPPAAL.

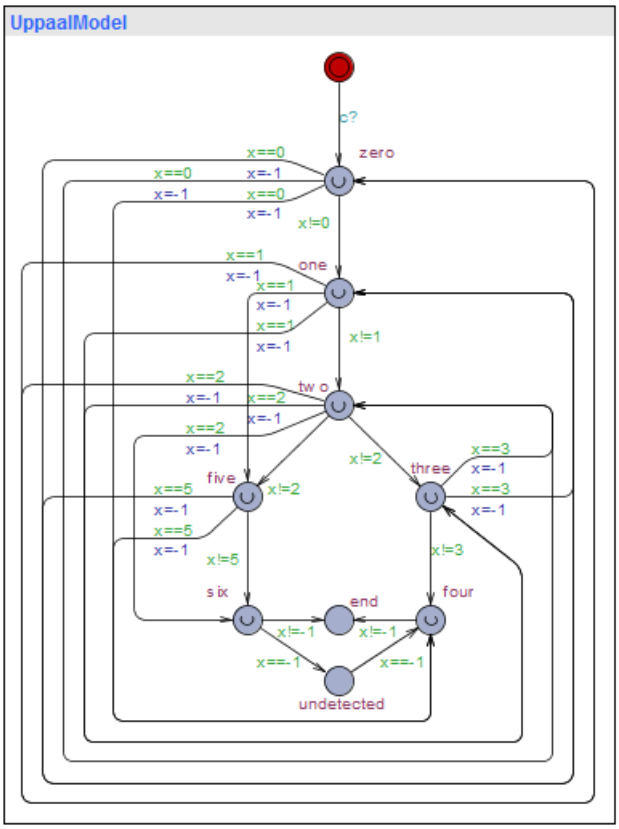


Figure 3.2: UPPAAL model with attack edges.

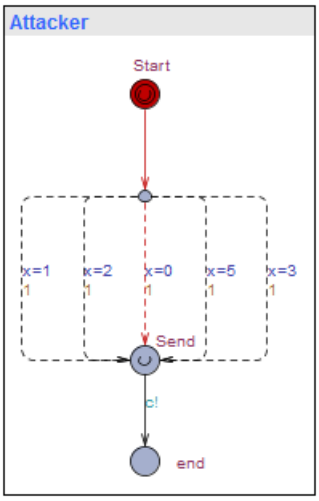


Figure 3.3: The attacker model.

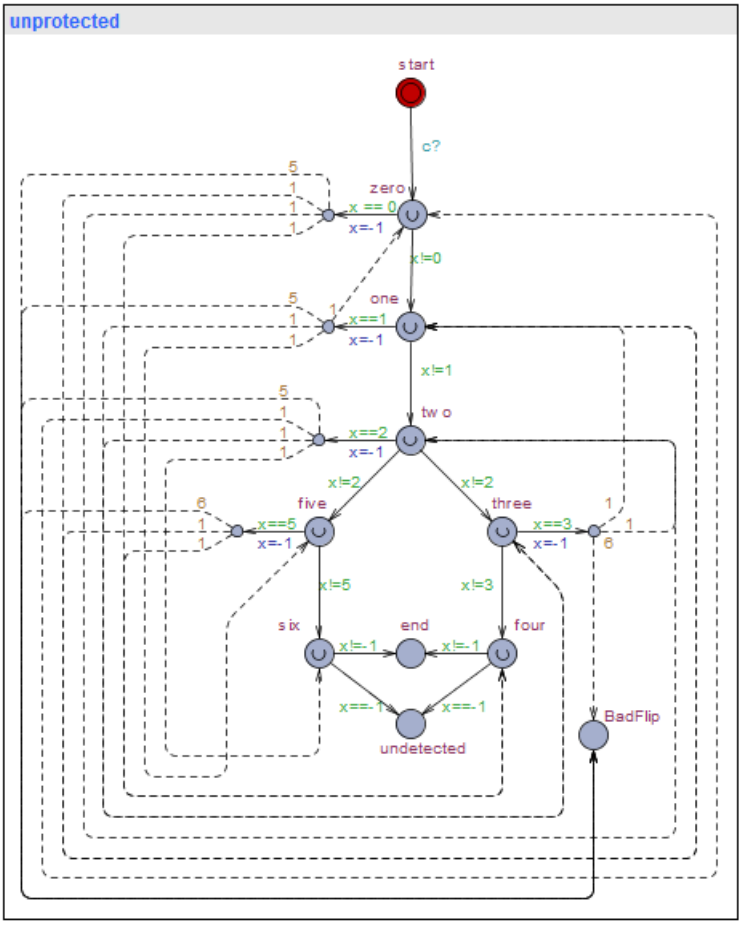


Figure 3.4: UPPAAL-SMC model with attack edges.

3.2 Modelling Programs

The following section will go into further details with what design decisions were made in choosing how to model programs in UPPAAL.

3.2.1 Modelling Start and End of Execution

To address an issue in the way we have chosen to model the attacker, it is necessary to create an 'artificial' hook-in into the model. This corresponds with the assumed program environment as shown in Figure 1.1, in the sense that program execution exists before and after the method that we analyse.

Essentially this is a state that can only be left by making a synchronization with the attacker on the only outgoing edge. This state is passed prior to entering the state of the first bytecode instruction in the program. This start state has been modelled as `start`. Finally, end of normal execution through the program has been modelled as `end`. This state is simply a way of concluding a correctly executed program where a bitflip has not occurred.

However, it is also possible to reach end of execution when a bitflip has occurred. This is what state `undetected` means, a bitflip has occured, it was not caught by the defensive measures introduced in Section 1.2, and end of execution has been met. Incidentally the `detected` state, is a state that is reachable only when a bitflip has occurred, and the defensive mechanisms has correctly acknowledged the attack.

3.2.2 Modelling Bad Bitflips

The maximum number of bytecode instruction states achievable in an 8-bit program is 256 (unsigned assumed). For this reason programs being modelled must only be 256 bytecode instructions long. This is a limit that has been set by us, to correctly model the number of bad bitflip transitions that may occur in a program. This limit also reflects exactly the limit on the amount of methods a Java Card applet may contain in a single class. Be advised however a Java Card package may contain up to 255 classes (not 256), each containing up to 256 methods[15].

A logical observation would then be, that if a program uses exactly 256 bytecode instruction states, each bitflip is guaranteed to hit another bytecode instruction - this corresponds with the notion of the bigger the target, the greater the odds of hitting something [12]. Should the program being modelled be shorter than the 256 bytecode instructions however, not all bitflip attacks leads to a bytecode instruction - but the PC register would still compute a valid value. In this case, a bad bitflip has occurred - a bitflip that is legal, but hits no correct bytecode instruction in a program. This attack will lead to `badflip` instead, as symbol for the undefined program behavior that occurs.

For illustrative purposes however, this is not modelled in the standard UPPAAL figures shown in this report and the SMC models' attack edges are weighted differently to compensate.

3.2.3 The Timing of a Bitflip

There are three ways to interpret, when a bitflip occurs. Does it occur before entering a given instruction state, does it occur during bytecode instruction execution or does it occur after entering a given instruction state. If the bitflip occurs before entering a given instruction state, it is simply meant that the bitflip edge can be chosen, before the bytecode instruction is executed.

If a bitflip may occur during bytecode instruction execution, it is meant that the PC may be manipulated during the execution of a bytecode instruction.

If a bitflip may only occur after entering an instruction state, it is meant that the bitflip occurs when the program state is about to be left for another, and thus the bytecode instruction in the state, has already been executed.

The choice of interpretation we have made in this report, is that the bytecode instructions will run atomically. For this reason all bitflips that occur, must occur after the bytecode instruction in a given state has already been executed.

3.2.4 The Number of Bitflip Attacks

As can be seen by the models introduced in Chapter 3, only one bitflip attack per program execution has been modelled. This is because of the observation that the defensive strategy in Section 1.2 does not seek to defend against higher-order bitflip attacks (multiple upsets), so testing the strategy against this kind of attack is not a correct observation of its defensive additions to a program.

3.2.5 Success Criteria of a Bitflip Attack

We have defined a succesful bitflip attack to be any bitflip transition chosen. To us, we do not discriminate if an attack ends up in a desirable position or an undesirable position - a succesful attack is one that changes the ordinary control flow of a system in some way.

Chapter 4

Modelling Java Card Defense Mechanisms

This chapter will model the defensive strategy that has been described in Section 1.2. This method of protection in particular is interesting, because it makes changes to the control flow graph of an application. There is a number of necessary changes to the operational semantic that must be done, to be able to implement the defensive strategy in these models. These changes will be shown in this chapter. The conclusion of this chapter is a new model implementing these security features.

4.1 Protective Measure

This section introduces the control flow of a program that employs validation by way of counters. A counter is either incremented or decremented and then compared against an expected value at certain points. See Section 1.2 for an explanation of this defensive strategy.

The example from the past chapter is the base, however as seen in Listing 4.1 the method has changed. The counter is passed with the method as a variable. The counter is decremented once at lines five through eight of Listing 4.1. As before, it is being compared against zero in lines nine through eleven, which is a predefined expected value to check against. The comparison is made with zero to open up to the possibility of optimization later, reducing the number of vertices needed, as explained in Section 4.2.

Adding these countermeasures also requires the introduction of two new endstates, which is the vertex 'detected' and the vertex 'undetected' as described in Section 3.2.1. These new endpoints allow us to distinguish between control flow attacks that have been caught before the program finishes its execution and those who successfully slip past these security measures. When determining the effectiveness of these countermeasures, the likelihood of a protected model ending in the vertex 'undetected' should be compared with the likelihood of

the unprotected model ending in the 'end' vertex. The end-state 'end' remains to catch those runs of the model who run to completion without reaching the vertex where the attack was supposed to happen.

Listing 4.1: The bytecode for the unoptimized example.

```
0  load 1
1  load 2
2  cmp= 9
3  push 0
4  return
5  load 3
6  push 1
7  math-
8  store 3
9  push 0
10 load 3
11 cmp= 13
12 exception
13 push 1
14 return
```

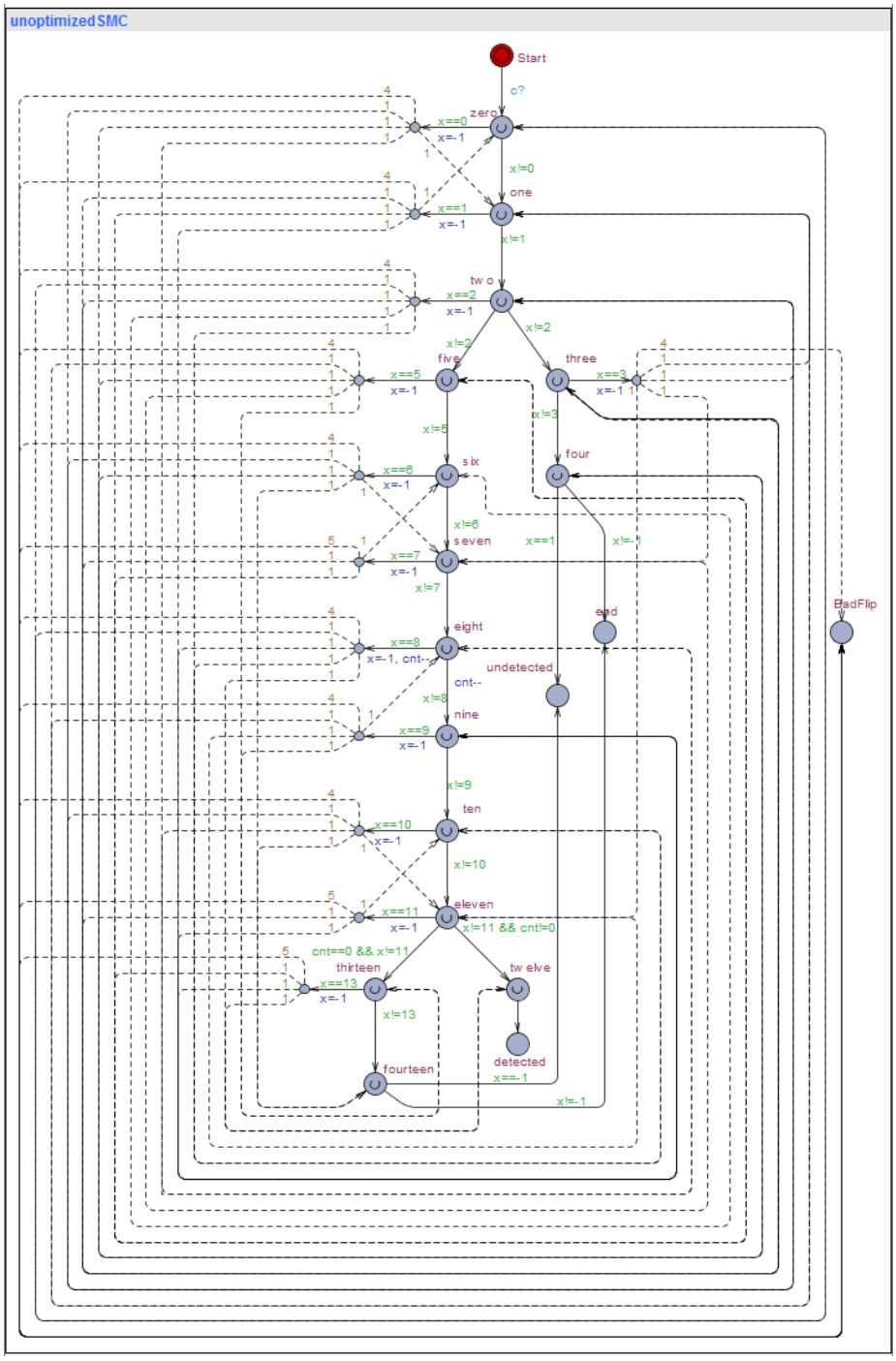


Figure 4.1: Unoptimized SMC Model.

4.2 Optimization of methods using Instructions

Having many instructions in a method may weaken its degree of security as described in Section 3.2.2. Essentially, more instructions in a program leaves less room to bitflip into a program state, that leads to a program crash (`badflip`).

This becomes a problem for our instruction set, because the correct implementation of this defensive strategy uses instructions that exists in the real JCML, but does not exist in our language. It is however possible for us to model it, but doing so would require loading the variables needed, and using math to decrement them and using store to save this variable change. Suddenly the decrement instruction in the JCML becomes 4-instructions long in our semantics. This is an incorrect representation of something that can potentially make the program more vulnerable. For this reason, it has been decided to introduce more instructions into the language we have constructed in order to reduce the number of instructions needed to perform certain high level tasks. The tasks we look to optimize are those needed to perform the security measures, that is to increment on a variable and to check if the variable has the expected value.

The instructions 'inc' and 'zcmp' have been defined based on the existing instructions 'iinc' and 'if<condition>' from the JCML the instructions. 'inc' takes a known variable and increases or decreases its value using a constant value, while 'zcmp' takes takes a value from the stack and sees if it is equal, lesser than or greater than zero. Finally 'exception' kills the execution of the program, it is a pseudo end state responsible for ending an illegal control flow when caught, but in it of itself does not possess any logic or rules.

Performing incrementing and comparing are four and three instruction respectively, but with the inclusion of these new instructions it can now be done with one and two instructions instead. When applied to the example method we can see how it shrinks down in Figure 4.2.

New Instructions inc and zcmp

inc is an instruction responsible for incrementing or decrementing a variable stored on the Local Variable Array. The instruction requires an Index value to select what Variable is being changed and a value to represent how much the Variable is being changed.

[INC]

$$\frac{instructionAt(m, PC) = inc\ i\ v}{P \vdash \langle H, \langle m, PC, OS, L\rangle :: FS\rangle \Rightarrow \langle H, \langle m, PC + 1, OS, L[i \mapsto L[i] + v]\rangle :: FS\rangle}$$

zcmp is an instruction responsible for comparing a value to Zero. Like a normal compare the instruction requires a Branch line number for method to jump through if the comparison returns true. If the comparison returns false it just increments the PC normally. In either case the value is popped from the Stack.

[ZCMP 1]

$$\frac{instructionAt(m, PC) = zcmp_{\bowtie}\ Br}{P \vdash \langle H, \langle m, PC, v :: OS, L\rangle :: FS\rangle \Rightarrow \langle H, \langle m, Br, OS, L\rangle :: FS\rangle}$$

$$Where\ v\ \bowtie\ 0$$
$$and\ \bowtie \in \{<, >, \geq, \leq, =, \neq\}$$

[ZCMP 2]

$$\frac{instructionAt(m, PC) = zcmp_{\bowtie}\ Br}{P \vdash \langle H, \langle m, PC, v :: OS, L\rangle :: FS\rangle \Rightarrow \langle H, \langle m, PC + 1, OS, L\rangle :: FS\rangle}$$

$$Where\ v\ not\ \bowtie\ 0$$
$$and\ \bowtie \in \{<, >, \geq, \leq, =, \neq\}$$

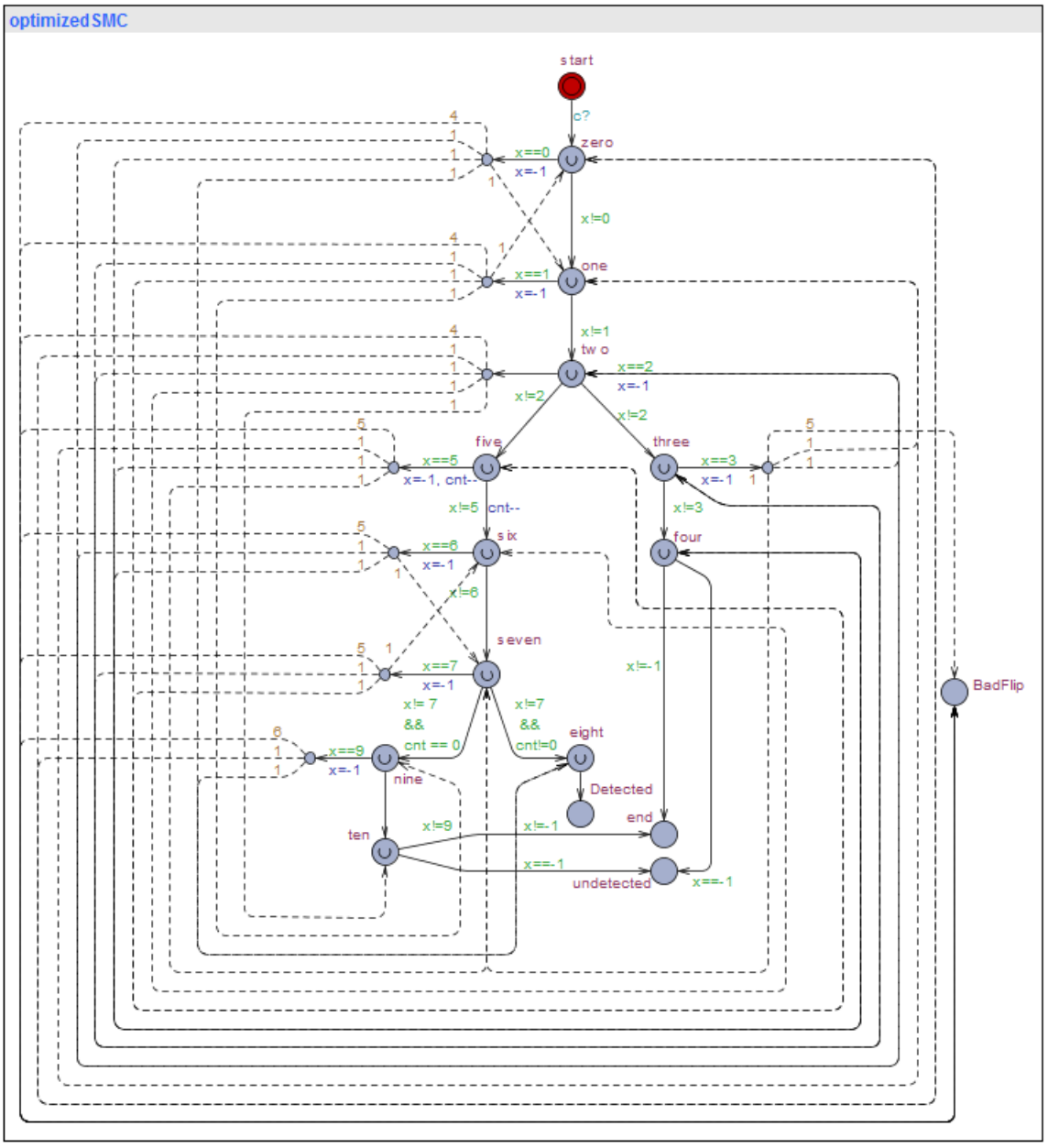


Figure 4.2: Optimized SMC Model.

Chapter 5

Program Analysis with UPPAAL

The work focus so far has been to make it possible to model a method in UPPAAL in a way that shows: a method's control flow, how performing a bitflip can affect that control flow, and how counters can be used to detect and end illegal control flows made by bitflips. This chapter will show how UPPAAL can be used to conduct program analysis.

5.1 Tool Description

There is no resource online from which we can extract programs that employs our JCVM language as defined in [18], and modeling programs in UPPAAL has proved to be time consuming process.

Because of this, two tools were created to do the preliminary work.

The two tools that were created are a code generator and a model parser.

5.1.1 Program Generator

The code generator combines modules of byte code instructions into methods that can be modeled later. The code modules contain also contains the tokens 'label', 'cntdec' and 'cntcheck'. The tokens are needed by the parser to apply attributes to the vertices. Labels are used to assign edges between a 'goto' instruction and its target, 'cntdec' assign which vertex should decrement the counter, and 'cntcheck' shows which compare should have guards to check the count.

After the method has been generated by putting together a series of code modules, it is then balanced, to make sure that all possible paths through the method contains the same number of decrements to the counter.

The tool then makes three copies of the method: One fleshes out the countermeasures by following the tokens with the instructions they need to have to per-

form the assigned task. The token 'cntstart' is added to the top of the method, complemented with the number of count decrements a path, is added to the start of the method along with an 'inc' to instantiate the counter in the method. The second method removes the tokens, except label, but adds 'NOP' instructions ("No OPeration") where the countermeasure instructions would have been. The last version simply has the tokens removed, again except label tokens.

Note that the code generator is programmed to take a maximum depth. Maximum depth is the maximum number of layers a formation can add to the code. For instance the noReturn formation with a max depth of five can at most expand the code with "instVar instVar 'math' 'store' noReturn" five times before it is forced to chose "instVar instVar 'math' 'store' " to stop adding more layers. This feature was added to give a rough control over how big a method can be, even if the size will still be largely random.

The formation rules that the tool uses to create the methods can be seen in Table 5.1. All the code is saved in text documents.

The algorithm responsible for balancing the code before can be seen in Listing 5.1. For the code to be balanced, all paths through a method must reach the same number of decrements to the counter before ending. A path is defined the succession of instructions that is reached throughout the execution of a method and there are potentially many unique paths through a method as, as compares will result in different instructions being reached depending on whether the comparison is true or false.

In order to assure that all paths are balanced the algorithm will recreate all possible paths though the method. This starts with a single path at the start of the method. When the path reaches a compare, a new path is created from the existing path, it has the same history as its parent but continues down the true branch where the parent will go down the false branch. The paths will continue, being produced whenever a new compare is reached, until it reaches the end of the method, where it will register the number of decrements it has encountered and return.

If there are paths that have encountered more decrements than other paths, the algorithm will balance itself by adding decrements to the end of paths who come out short.

There is a weakness with the algorithm in its current form, it cannot handle instances where multiple paths end at the same point, but have encountered a different number of decrements. It can currently only handle cases where paths of a distinct number of decrements do not reconnect. However, the code generator does not generate methods where this would be a problem as it is not possible with the formation rules.

Listing 5.1: The pseudocode outlining the method used to balance.

```
0 Preparation:
1   Create a queue.
2   Create a 'Path' Containing
3     A count, for the number of cntdecs in the flows path, in↩
```

```
Code::= ('cntstart' 'inc' || 'NOP' || ' ') withReturn

ifblock::= instVar 'label#2' instVar instVar 'cmp label#1' withReturn
 'label#1' noReturn 'goto label#2' ||

 instVar instVar 'cmp label#1' cntdec 'label#2'
 withReturn 'label#1' cntdec noReturn 'goto label#2' ||

 instVar instVar 'cmp label#1' cntdec withReturn
 'label#2' withReturn label#1 cntdec noReturn 'goto label#2'||

 instVar instVar 'cmp label#1' cntdec withReturn
 'label#1' cntdec withReturn

noReturn::= instVar instVar 'math' 'store' ||
instVar instVar 'math' 'store' noReturn

withReturn::= cntcheck 'return' ||
instVar instVar 'math' 'store' withReturn ||
ifblock

cntdec::= ('cntcheck' 'inc' || 'NOP' || ' ')

cntcheck::= ('load' 'cntcheck' 'zcmp label#1' 'exception' 'label#1' ||
'NOP' 'NOP' 'NOP' || ' ')

instVar::= 'load' || 'push'
```

Table 5.1: Formation rules for the code generator.

```
         the at zero,
    A line count at zero (the first line),
    An empty list history of previously visited 'cmp' instructions,
    A reference to the code being checked.

Step One:
  Is the queue empty? If not:
    Dequeue the top Path and examine the instruction it is at in step Two.

  If it is:
    Continue to step Three.

Step Two:
  Is the instruction 'cntdec'? If so:
    Increment the count and the line count. Add the updated Path to the queue.

  Is the the instruction a goto? If so:
    Update the Path's line number to be that which the goto's target label is at. Add the updated Path to the queue.

  Is the instruction a compare? If so:
    Has that particular compare been visited before? If so:
      Add the compare instruction to the history list.
      Create a new Path by copying the current one.
      Update the new Path's line number to be that of the label that marks the start of the true branch. Add the path to the queue.
    Update the (original) flow unit to go down the false branch and add it to the queue.

  Is the instruction a 'cntcheck'? If so:
    This is the end of a flow units path, append the path's count to the cntcheck's line of code.

  Is the instruction 'anything else'? If so:
    Increment the line count.

  Is the flow units current count higher than the global max count? If so:
    Update the global max count.

  Continue from step one.

Step Three:
  For every line in the code being checked, does the line contain a count appended to it? If so:
```

```
    Is the count smaller than the global maximum? If so:
      Add new lines with the token 'cntdec' before the line ↩
          with the count appended.
    Remove all appended counts from that line.
```

5.1.2 Model Parser

The parser reads the generated programs from a .txt format, and generates an UPPAAL-ready XML document. This document is essentially the model. The instructions of the program are translated to vertices and the parser creates the edges between them based on the control flow established by the instructions. Potential bitflips are calculated and connected to instructions or the badflip state by the parser. Special rules associated with the counter measures are added to vertices that follow the related tokens, in the form of guards and updates. These special rules include guards to limit movement around count checks, and updates to simulate the count being decremented.

For a full overview of the parser, see Appendix A and Figure A.1.

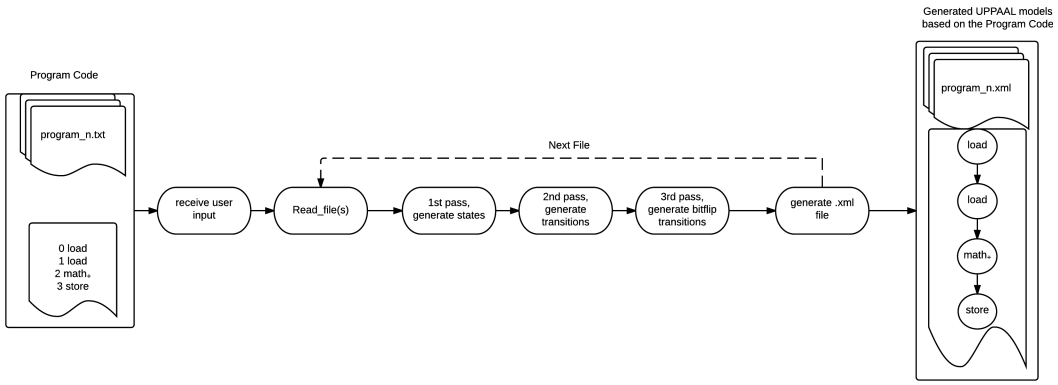


Figure 5.1: Program Flow of the Parser.

For the purpose of better understanding of the mechanisms in each step, each figure illustrates by way of plain-text and a flow chart the algorithmic approach behind them. Not all flags are described in the figures, for example the `x != <constant` flag, because it is a property that every edge has so it would not make sense to show. Only the important portions of the algorithms have been highlighted in the following figures.

Figure 5.1 illustrates the program flow of the Model Parser. The main thing to understand from this figure is that the parser goes through three-steps for each program file, with each step contributing something to the CFG of the program. The first step is illustrated in Figure 5.2, and simply creates the necessary UPPAAL states. This includes the end-states `badflip, undetected, detected` and `end`. The second step consists of two steps as illustrated in Figure 5.4. The first step (2a) is responsible for connecting states with edges, including edges

between bytecode instructions and the relevant end-states. This step is followed by a miniature step. This step (2b) is responsible for connecting the remaining edges from a `comparison`'s true-branch to the remaining program instructions. This is also a step wherein `goto`'s are connected to the correct bytecode instructions in the CFG. The final step is illustrated in Figure 5.3 here bitflip transitions are connected to the correct bytecode instruction states, as well as the `badflip` state.

Program Details

The parser was written in C++, which means that a ready to use executable file can be produced. The parser behaves reasonably in terms of performance as well when generating models, even though as illustrated in the figures, many passes over the same program-file is conducted. The program however as designed as a proof of concept rather than an off-the-shelf solution to be used by the industry at large, and as such the 'prettiness' of the models was something that was not prioritized, neither was the program's design both in regards to object orientation but also exception handling. For the parser to work it is necessary to keep the `parser.exe` in a folder with two subfolders, `uppaal_program` and `uppaal_parsed`. Furthermore, the command prompt should only be used with the intended input. The Parser can construct `UPPAAL`, `UPPAAL SMC` and `UPPAAL TIGA` models of the same program(s).

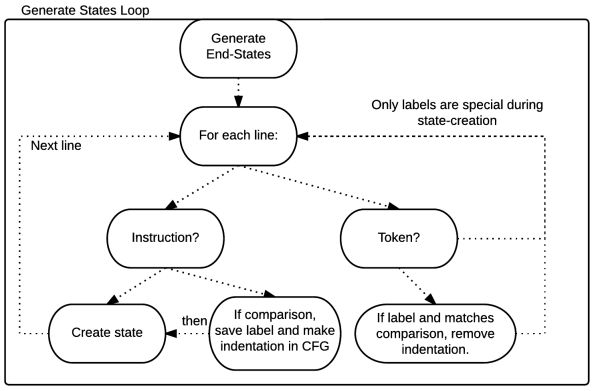


Figure 5.2: Step 1, Generate States.

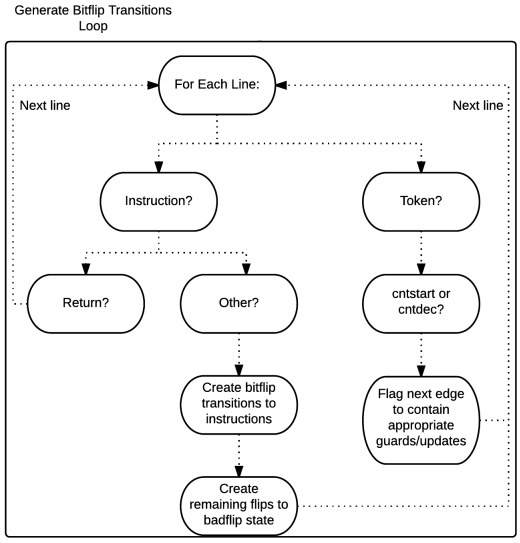
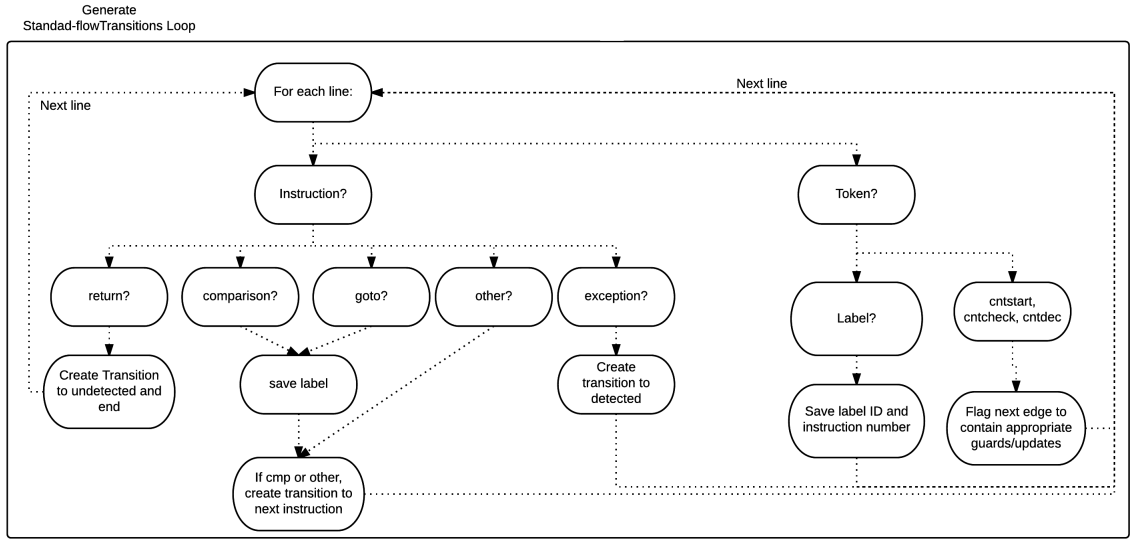
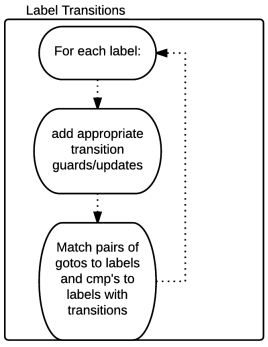


Figure 5.3: Step 3, Generate Bitflip Transitions.



(a) Step 2a: Generate Standard Transitions.



(b) Step 2b: Add Remaining Label Transitions.

Figure 5.4: Step 2, Generate Transitions.

5.2 Queries and Evaluation

The following section will describe some of the ways that UPPAAL allow us to analyze our models. We have decided to conduct experiments within three branches, namely Balancing, Execution Flow and Distribution.

5.2.1 Balancing

The balance-test is performed by bypassing the attacker so that no attack is performed on the model and then using the queries "$E <> Program.detected$" and "$E <> Program.undetected$" to see if it is possible to end a run in end-states normally reserved to attacks.

To bypass the attacker, the parser (in SMC and UPPAAL) guards the edge to the branchpoint with a variable `testbalance`, and produces a new edge from `start` to `send` that does not manipulate `x`, this transition is also guarded. It is then possible in declarations to decide what 'mode' to run the model in, assigning 0 to `testbalance` (as is the default setting), means that the model will run 'normally'. Assigning 1 to `testbalance` means that the attacker will be bypassed and it can then be used for balance testing models. This can be seen in Figure 5.5.

In cases where the model is imbalanced, UPPAAL can generate a trace that shows exactly what path is imbalanced. The query "$E <> Program.detected$" is used in this example. This trace is a sequence of instructions that need to happen for the run to end in the `detected` end state. The developer is then able to follow the imbalanced sequence and compare it to the bytecode instructions in order to discover the problem.

This method was used to discover a bug in the implementation of the balancing algorithm of the code generator. After generating the code and during the balancing step the program would inject count decrements one step too early in the raw code. On rare occasions where a count check would immediately follow a label, the injected count decrements would land before the label, and would not correct the imbalanced path but would rather alter a different one, if any. This error has been corrected, the effect it had and the trace that UPPAAL generated can be seen in Figure 5.6 where line 27 is placed before the label and becomes dead code that is never accessed, rather than being after the label where it would have balanced the path.

This goes to show that UPPAAL can be used for path balance testing as well.

5.2.2 Execution Flow

Execution Flow tests in this report tests reachability and trace based properties in each model. What is interesting to know in this regard, is up to the analyst of course, however to start us off we will ask certain queries about the integrity of a program. For illustrative purposes we will use the model in Figure 4.2.

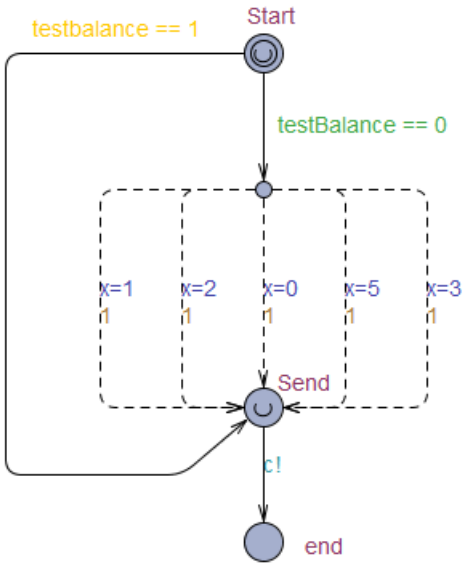


Figure 5.5: Circumventing the attacker to check the model balance.

Bypassing Protective Measures

A query that will allow us to test whether it is possible to bypass the defense mechanism in instruction 7, can be asked by `E<>optimizedSMC.undetected`. This property is satisfied, by performing a bitflip transition in `zero` to end up in state `four`. This is a fairly sterile test-case though in the sense that this kind of analysis does not take into account the chance of hitting a wrong state. This query can thus be expanded, and instead it is possible to ask what is the probability, that this state is reached. This kind of result is produced as a confidence interval as described in Section 2.3.1. This query is: `Pr[<=50](<>optimizedSMC.undetected)`. The result shows that with 95%, the probabilistic value of reaching `undetected` is somewhere between 18% and 28%.

5.2.3 Distribution

For balanced models, the developer can use UPPAAL to determine how much security is afforded to the method he has programmed. Security is defined as the countermeasures added to the method, and how likely they are to detect an attack. To determine this UPPAAL can be used to display how a series of runs are distributed along the end points, based on the model randomly chosen attacks and where the attacks reach. To know where the runs end the edges to every end-point in the model has updates a value 's'. When the run reaches `detected`, s becomes 1, in `undetected` s becomes 2, in `badflip` s is 3, and in

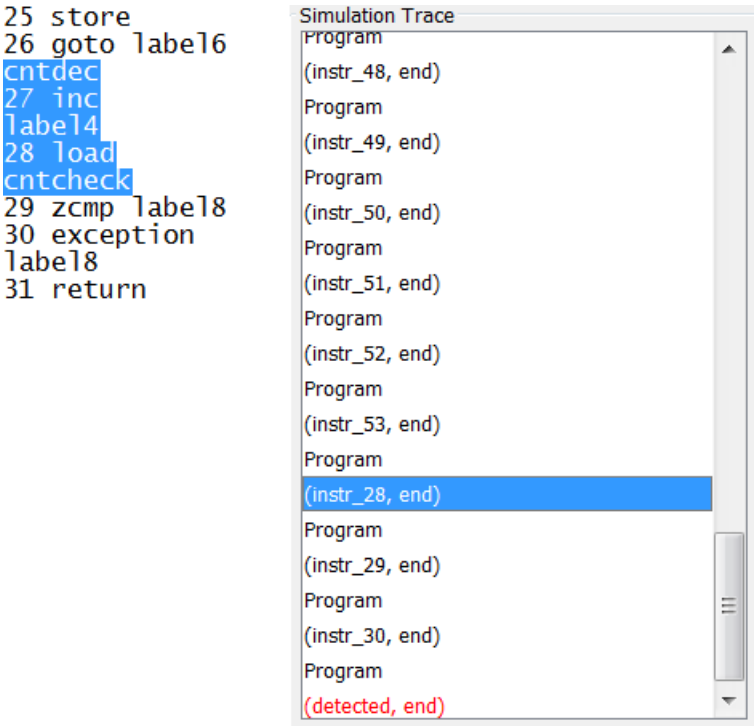


Figure 5.6: The misplaced count decrement.

`end` s becomes 4.

- 'Detected' is the number of runs that make an attack and is caught the counter measures, not applicable for NOP- and Unprotected versions.
- 'Undetected' is the number of times where an attack has happened, and is not detected by any counter measures.
- 'BadFlip' is the number of times an attack is attempted but the PC lands out of bounds.
- 'end' is the number of times a run completes without performing an attack, this occurs when a run chooses a path down one branch of a compare, and the attack was supposed to happen at the other branch.

The query"$E[\Leftarrow 50; 20000](max : s)$" is then used to find the distribution of the runs. The query returns the evaluation of twenty-thousand simulated runs with a time bound of fifty. This is illustrated in Figure 5.7 where the query displays how many times a run ended with a given value for 's'.

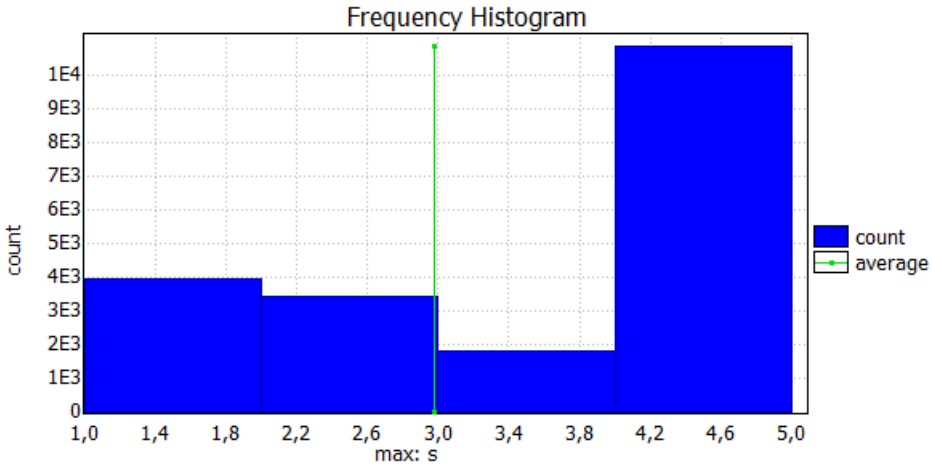


Figure 5.7: Query results as displayed in UPPAAL, where s values represent 1: Detected, 2: Undetected, 3: Bad Flip, 4: End.

To exemplify how the developer could determine this we have generated five methods with the code generator. There are three versions of each method: Unprotected is the method with no counter measures, while protected is the method with countermeasures. The NOP version is an unprotected method created by replacing all countermeasure instructions in the protected version with 'No Operation' instructions - this gives us a program of equal length with the protected version. The Table 5.2 shows the results of the distribution queries on the different models. While averaging the method versions on end shows that between 7.26% and 65.5% of the runs end without performing an attack, the

result of the query also shows that adding the counter measures reduces the number of undetected attacks at an average of 35.34% when compared to the NOP version.

This shows us that there is an impact in adding counter measures to a program. It appears that counter-measures takes runs from `undetected`, but also quite possibly `end` or `badflip`. Unfortunately, because of the way the current model works, it is not possible to estimate how much of an impact counter measures have, it is only possible to say that there is an impact.

Name	Line Count	Detected	Undetected	BadFlip	End
1-Unprotected	33	N/A	10306	8115	1579
1-Protected	38	1256	9671	7671	1402
1-NOP Version	38	N/A	10505	8115	1380
2-Unprotected	70	N/A	8212	3714	8074
2-Protected	78	3131	5742	3670	7457
2-NOP Version	78	N/A	8723	3798	7479
3-Unprotected	91	N/A	6490	2690	10820
3-Protected	102	1911	5126	2413	10550
3-NOP Version	102	N/A	6771	2658	10571
4-Unprotected	151	N/A	9123	2380	8497
4-Protected	166	4814	4703	2097	8386
4-NOP Version	166	N/A	9302	2247	8451
5-Unprotected	162	N/A	5673	1459	12868
5-Protected	202	3415	2506	830	13249
5-NOP Version	202	N/A	5801	1026	13173

Table 5.2: The distribution of runs on the example methods.

Using the query the developer is capable of determining if the introduction of countermeasures to the method are effective. Alternatively they can compare versions of the same method, where the countermeasures are placed differently, to determine which implementation of the counters provide the method with the highest likelihood of detecting an attack to the control flow. It is interesting to note that even from the small number of tested programs, there is an apparent correlation between the number of attacks that are foiled by ending in "badflip" and the size of the method. The number of runs ending in "badflip" decreases as the number of lines increase.

Chapter 6

Reflection

The following chapter will reflect on the project. The chapter will consist of a future work portion, which will highlight how the toolchain might change or be used in the future. A look at the modelling technique chosen for this project will also be discussed. The last part of the chapter will be a conclusion, that seeks to conclude on the project based on the problem definition given in Section 1.3.1.

6.1 Toolchain Evaluation

The following section will evaluate the toolchain that was programmed throughout this semester and seek to highlight areas of improvement. The program generator was a necessary step in creating programs that were not made by hand, however beyond this project it is difficult to see its use in the future, this will be explained in the following.

6.1.1 Model Parser

The next step for the model parser, is a re-implementation to flush out bugs and produce a more efficient program-design. There are errors that we are aware of, but have chosen not to fix, because these bugs does not at all impact the output that is the UPPAAL ready control flow graph. Furthermore, it would be interesting to make the necessary tweaks to be able to support the full Java Card Machine Language. As of right now the model parser may only parse programs written in our operational semantics.

In their current form the model parser and the program generator should simply be considered a proof-of-concept that illustrates how simple mediation software is able to transform program-code into UPPAAL models. The parser currently supports three UPPAAL versions (UPPAAL, UPPAAL SMC and UPPAAL TIGA) but this is also an area that could be expanded, and with proper re-implementation, could be a task not too challenging. On a side-note we discovered that UPPAAL Stratego is fully supported with our models, this is

because the coding conventions of UPPAAL in later models seems to align, whereas TIGA is clearly an older version that does not support new .xml features - so it is not a necessary given that dramatic changes to the output of the parser has to be made. This is because the parser outputs .xml models for UPPAAL's latest edition (4.1.19)[4].

The parser is reasonably scalable as is, but proper use of Object Oriented design would make this task easier and more flexible.

Parser Scalability

Should the parser have to be scaled up to the full JCVML, the task would not be infeasible. This is because general rules for each vertex can easily be established. At the correct level of abstraction, the difference between `store` and `load` is non-existent at least from a control flow graph perspective. The tricky parts for the parser, is correctly connecting jumps in branches. There are many more jump statements and if-statements in the full language however, and a degree of abstraction between these overlapping instructions as suggested in our operational semantics could be used with the same benefits.

Changes to the Model

As suggested in Section 6.3, to be able to model the impact of a guaranteed bitflip in the model, the parser will need to work differently. A solution we would propose is that the parser builds a tree data structure of instructions, as opposed to reading line-after-line as is. With this design it will become a more trivial task to know, which instructions is found in each branch. When a comparison instruction is then encountered, it is simply a matter of traversing the tree following the comparison statement, and construct a guard based on the instruction names as illustrated in Figure 6.1.

6.1.2 Program Generator

The program generator was a tool of necessity written to construct programs that employed the defensive mechanism described throughout this report. This is because we do not have access to a codebase to draw from, so to be able to construct our own programs was the only solution to be able to conduct experiments of reasonable size and objectivity. There are issues though with this approach, for example it is possible that the programs, because of the small language in the semantics, are very much the same - if not semantically, then control flow graph wise at least. Furthermore, the ways in which we place the defensive mechanisms in the programs generated is not based on any best practices available within the industry, rather they are placed algorithmically, based on examples from [1].

6.2 Alternative UPPAAL Models

Instead of having conducted this analysis in SMC, another approach could have been to use UPPAAL Stratego and UPPAAL TIGA. UPPAAL Stratego is a fairly new version (first released in April 2015[7]) of UPPAAL which combines SMC's flexible nondeterminism, with TIGA - essentially making it possible to create more complicated games than in just TIGA. TIGA is a strong tool for synthesizing and visualizing strategies in timed game automata. The following section will go into details with these two UPPAAL versions and describe their use for this project.

6.2.1 TIGA and Stratego

A big difference between TIGA and Stratego is that Stratego makes sacrifices when it comes to visualizing the strategies that has been synthesized in a model, which is where TIGA comes into its own from an analytical point of view. In return however, Stratego allows for more complex models, by combining UPPAAL SMC and its probability weighting as well as query language, with Stratego's own, this is something TIGA does not allow. Essentially, a UPPAAL SMC model may be used directly within Stratego, whereas a SMC model must be entirely re-written, to fit with TIGA.

Strategy Synthesis and Further Analysis

In Stratego, strategies can be assigned a special variable by expanding TIGA's strategy syntax. A practical example could be: `strategy S = control: A<>Program.undetected`, in which case the winning condition is reaching the undetected state in Program[1]. This strategy can be utilized with UPPAAL SMC however, for example with: `Pr[<=2](<>Program.four) under S`, in which case the query seeks to know the confidence interval with which the `undetected` state can be reached from state `four`. If a similar query is asked in TIGA control: A<>Program.undetected, the output if satisfied, is a path in the UPPAAL simulator which illustrates the necessary transitions to satisfy the query. Furthermore a strategy can be given through the command prompt application accompanying TIGA, that shows the optimal transitions to satisfy the query, in every state - both things Stratego cannot do[8].

Synthesizing strategies in the way of TIGA and the simulator is quite interesting, because TIGA is able to show which states lead to the query being satisfied. If the query control: A<>Program.undetected cannot be satisfied in one of our models, it means that there is no guaranteed winning strategy to always reach this state. If it can be satisfied however, the diagnostic trace will reveal which path allows this, and potentially uncover points of weakness in the model that could be remodelled, or simply removed from the control flow.

[1] During strategy synthesis, the computer takes on the role of the attacker (bitflip transitions), and the user is the defender (program)

6.2.2 Preliminary Results

Unfortunately for this project, we were not able to fully utilize TIGA (due to time constraints) or Stratego (discovered the tool too late in development).

The preliminary results we found, were based on an ad-hoc program analysis approach after running queries of the kind mentioned in TIGA with the query: `control: A<>Program.undetected`. Each model generated by the program generator produced simply one flow with which it was possible to always enter either the undetected state in a model. The program was changed to try and eliminate this weakness, and though TIGA suggested that now no winning strategies existed, testing the same model in SMC proved that it was indeed still possible to enter the undetected state, albeit less likely. UPPAAL SMC is not an optimal 'adversary' that makes optimal choices in every state, rather the stochastic approach of SMC means that as could be expected some runs will always reach the undetected state so long as there is a transition that leads to it.

Furthermore, this decrease in likelihood could come from numerous factors, for example that the program simply became smaller - recall that in theory a smaller program makes it more likely to make a bad bitflip transition because the target is smaller. Alternatively, perhaps the approach worked, and we were on to something. Or finally, perhaps the statistics that were drawn from this model were simply lucky to suggest a trend - or perhaps the generated program is at fault. It is too early to say, but it leaves room for further and more structured analysis in the future.

6.3 Changes to the Model

In the current model, there is a subtle problem with the `end` state. It is a state that is reachable when no bitflip attack has been made. To be able to model the outcome of a guaranteed bitflip attack, we came up with two potential solutions. The first was to create a path from `end` to `zero` to force the model to produce one or more runs. Unfortunately this produces a phenomenon known as zeno behavior, which can be considered to be sort of a livelock (in UPPAAL zeno behaviour means that an arbitrary number of transitions can be taken in 0 time units).

The problem this introduces is that the model can infinitely loop, and will be rejected by UPPAAL (because of its inherent zeno detection) - so this solution is unfortunately not valid.

The alternative solution however, is a bit more complicated, and in the current version of the parser cannot easily be implemented (which is why it was not done). The solution will however solve the problem, and guarantee that a bitflip occurs, in every run. This solution seeks to make the creation of `comparison` vertices and edges more complicated. This solution is illustrated in Figure 6.1.

The edges that deviate from the current model, are the branching edges. As

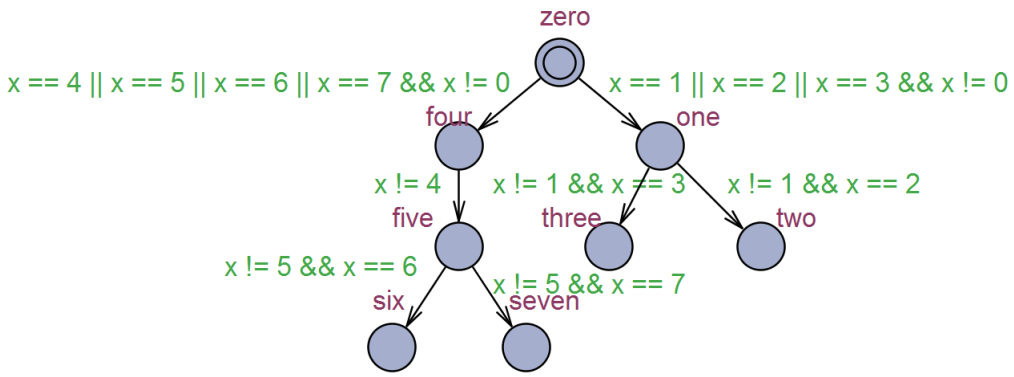


Figure 6.1: The future model.

Figure 6.1 shows, it is necessary to know the state-names of all future states in each branch, to be able to correctly funnel a path toward the state it must bitflip from. This makes the guards outgoing from the first state (`zero`), the most complex. After that, for each branch, the guards would become less and less complex. On normal edges (edge between four and five in Figure 6.1), the current `x!=<current state` rule will suffice.

6.4 Conclusion

The report has shown how it is possible to model Java Card applications in UPPAAL. A parser was written as proof of concept to show that this task can also be automatized. Because there does not exist any programs that is based upon operational semantics defined in [18], we wrote programs ourselves to work with the parser - this process was also automatized. Put together, we showed how it is possible to analyze a program of this kind.

It would seem that an attacker faces many challenges, when they try to tamper with the control flow of such an application. Assuming the card does not break when it is struck by the energy particles of a laser, the energy particles will also need to strike the circuit in such a way that something meaningful can come out of it. As we have shown in this report, the odds of making a bitflip, that does absolutely nothing (bitflips that goes to `badflip`) is higher for small programs, than bigger programs. This illustrates the notion of the bigger the target, the easier to hit [12]. A good coding practice for Java Card Applications and in general good object oriented design, is therefore to keep methods small. In case a bitflip attack would then occur, the odds of making a bad bitflip is increased.

As described in Section 6.3, there is an issue with the current model in the sense that not every run leads to an end state, where a bitflip has been made. In the current model however it is possible to benchmark how many of these runs simply goes to `end`. In an unscientific way, this state could potentially model

the runs which caused damage to the circuit and broke it. It is important to note that it is not a given that the way of modeling programs as suggested in Section 6.3, is a realistic approach - in fact it might be too pessimistic with the succesrate of a bitflip attack.

Bibliography

[1] Mehdi-Laurent Akkar, Louis Goubin, and Olivier Ly. *Automatic Integration of Counter-Measures Against Fault Injection Attacks*. URL: `http://www.labri.fr/perso/ly/publications/cfed.pdf`, 2003.

[2] Hagai Bar-El. Known attacks against smart cards. http://www.discretix.com/white-papers/known-attacks-against-smartcards/.

[3] Guillaume Barbu, Philippe Andouard, and Christophe Giraud. *Dynamic Fault Injection Countermeasure*, 2013. See Repository for PDF source.

[4] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *Uppaal tutorial*. See repository for PDF source.

[5] Boost. *Boost Library Website*. URL: `http://www.boost.org/`, 2015.

[6] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined software and hardware attacks on the java card control flow. *Smart Card Research and Advanced Applications: 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers*, pages 283–296, 2011. See Repository for PDF source.

[7] Alexandre David, Peter Gjøl Jensen, Kim G. Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. *Stratego Homepage*. URL: `http://people.cs.aau.dk/~marius/stratego/index.html`, 2015.

[8] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikucionis, and Jakob Haahr Taankvist. Uppaal stratego. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 206–211, 2015.

[9] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, pages 1–19, 2015. See Repository for PDF source.

[10] Bar-El H., Discretix Technol. Ltd Rehovot Israel, Choukri H., Naccache D., Tunstall Michael, and Whelan C. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370 – 382, Feb 2006.

[11] Allan. H. Johnston. Scaling and technology issues for soft error rates. *Presented at the 4th Annual Research Conference on Reliability, Stanford University, October 2000*, pages 1–9, 2000. See Repository for PDF source.

[12] Julien Lancia. Java card combined attacks with localization-agnostic fault injection. *Smart Card Research and Advanced Applications: 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, pages 31–45, 2013. See Repository for PDF source.

[13] Sun Microsystems. *The Java Card 3 Platform*, August 2008. `http://www.oracle.com/technetwork/java/javacard3-whitepaper-149761.pdf`.

[14] Wojciech Mostowski and Erik Poll. Malicious code on java card smartcards: Attacks and countermeasures. *8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008, London, UK, September 8-11, 2008. Proceedings*, pages 1–16, 2008. See Repository for PDF source.

[15] Oracle. *Java Card 3: Classic Functionality Gets a Connectivity Boost*, 2009. `http://www.oracle.com/technetwork/articles/javase/javacard3-142122.html`.

[16] Oracle. *Java Card 2.2.1 Platform, Virtual Machine Specification*, October 2003. See repository for PDF source.

[17] Tezzaron Semiconductor. *Soft Errors in Electronic Memory – A White Paper*, January 2004.

[18] Anders Vinther and Jakob Jørgensen. *Operational Semantics of a JCVM Language*. URL: `http://projekter.aau.dk/projekter/da/studentthesis/operational-semantics-of-a-jcvm-language(1a79dc0e-7f16-4dde-b97f-d929bb3c0f17).html`, 2014. See repository for PDF source.

Part I

Appendices

Appendix A

Model Parser

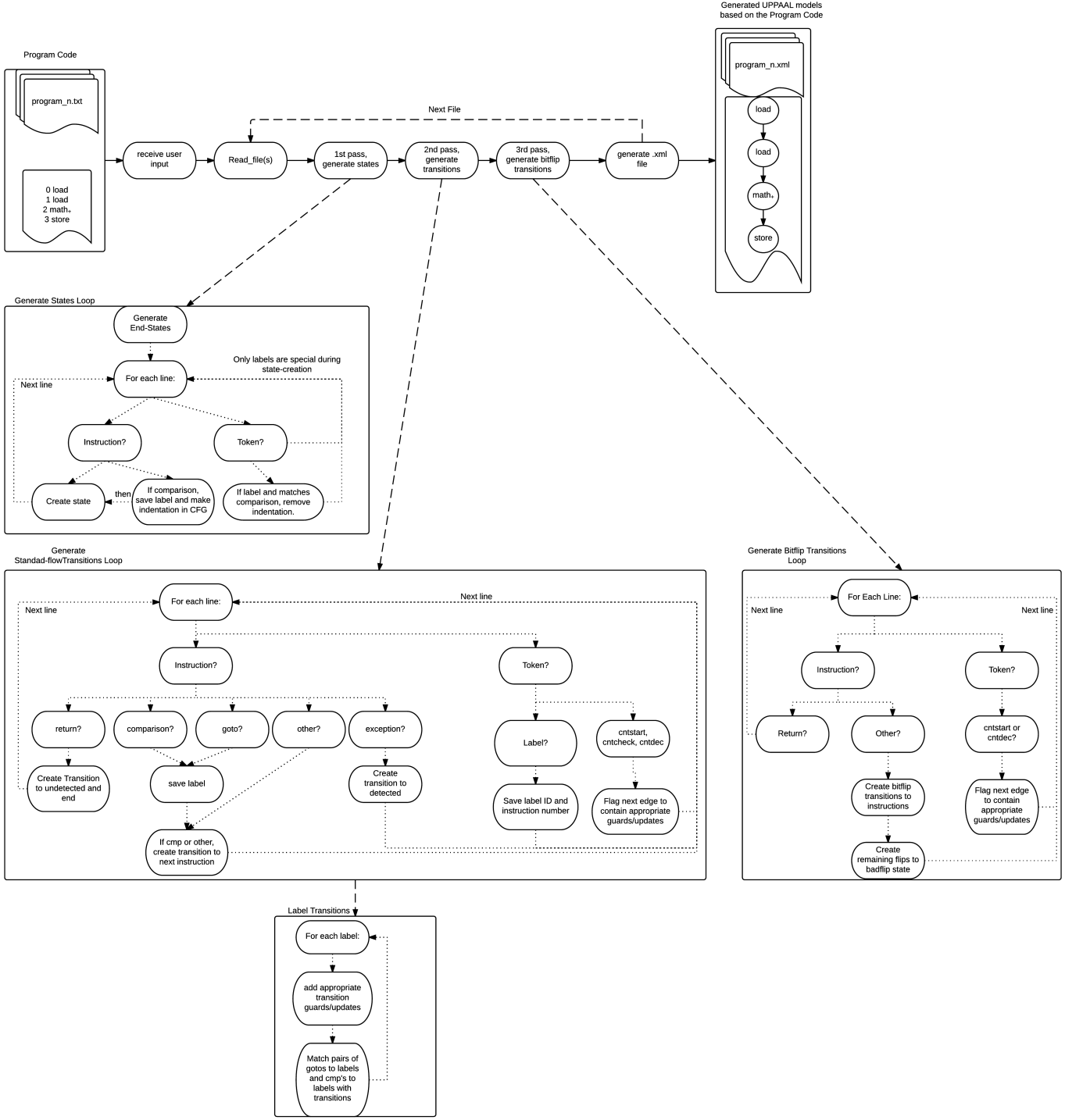


Figure A.1: Full Overview of the Parser.

Appendix B

Instructions of the Operational Semantics

This paper uses many instructions that have been defined in [18]. The rules define the transition from one configuration of a program P as shown: $P \vdash \langle H, \langle m, PC, v :: OS, L\rangle :: FS\rangle$. The transition itself requires some conditions to be true before the old configuration can be changed to the result of the instructions execution.
[NAME]

$$\frac{Conditions}{P \vdash Configurationbefore Execution \Rightarrow Configurationafter Execution}$$

The rules are cited from [18].

pop removes a value from the Operand Stack of the current method's Frame. It requires that there is a value on the Operand Stack. The method affects only the Operand Stack and the PC of the current Frame.
[POP]

$$\frac{instructionAt(m, PC) = pop}{P \vdash \langle H, \langle m, PC, v_0 :: OS, L\rangle :: FS\rangle \Rightarrow \langle H, \langle m, PC + 1, OS, L\rangle :: FS\rangle}$$

push adds the accompanying byte value to the Operand Stack of the current method's Frame.
[PUSH]

$$\frac{instructionAt(m, PC) = push\ v_0}{P \vdash \langle H, \langle m, PC, OS, L\rangle :: FS\rangle \Rightarrow \langle H, \langle m, PC + 1, v_0 :: OS, L\rangle :: FS\rangle}$$

Math is a simplification of all mathematical instructions. It affects the current Method Frame and requires two values that are not object references to be pushed onto the Operand Stack.

Both values are popped from the Operand Stack and math is performed on

them. After that, the result is pushed back onto the Operand Stack and the PC is incremented.

[MATH]

$$\frac{instructionAt(m, PC) = math_{\diamondsuit}}{P \vdash \langle H, \langle m, PC, v_0 :: v_1 :: OS, L\rangle :: FS\rangle \Rightarrow \langle H, \langle m, PC + 1, v :: OS, L\rangle :: FS\rangle}$$

$$Where\ v = v_1 \diamondsuit v\ and\ \diamondsuit \in \{+, -, \cdot, \div\}$$

cmp is the instruction that compares two values relative to some logic symbol. Because this expression can either be true or false there are two transitions for this instruction. The instruction requires an accommodating byte that is the branching value as well as two values on the Operand Stack (the values to compare). It only affects the current method Frame.

Version 1 is the transition where the comparison evaluates to true. Both values are popped from the Operand Stack and compared, since they are true the PC is set to be the branch value.

Version 2 is the transition where the comparison evaluates to false. Both values are popped from the Operand Stack and compared, since they are not equal to each other, the PC is incremented to move past the cmp instruction.

[CMP 1]

$$\frac{instructionAt(m, PC) = cmp_{\bowtie}\ Br}{P \vdash \langle H, \langle m, PC, v_0 :: v_1 :: OS, L\rangle :: FS\rangle \Rightarrow \langle H, \langle m, Br, OS, L\rangle :: FS\rangle}$$

$$Where\ v_0\ \bowtie\ v_1$$
$$and\ \bowtie \in \{<, >, \geq, \leq, =, \neq\}$$

[CMP 2]

$$\frac{instructionAt(m, PC) = cmp_{\bowtie}\ Br}{P \vdash \langle H, \langle m, PC, v_0 :: v_1 :: OS, L\rangle :: FS\rangle \Rightarrow \langle H, \langle m, PC + 1, OS, L\rangle :: FS\rangle}$$

$$Where\ v_0\ not\ \bowtie\ v_1$$
$$and\ \bowtie \in \{<, >, \geq, \leq, =, \neq\}$$

goto changes the PC value of the current method frame to that of the byte that accompanied the instruction.

[GOTO]

$$\frac{instructionAt(m, PC) = goto\ PC'}{P \vdash \langle H, \langle m, PC, OS, L\rangle :: FS\rangle \Rightarrow \langle H, \langle m, PC', OS, L\rangle :: FS\rangle}$$

load loads a value from the local variable array. It requires an accompanying byte representing the index where the required value is located. The instruction only affects the current method Frame.

The instruction loads the value from the Local Variable Array at the index location and pushes it onto the Operand Stack. The PC is then incremented.

[LOAD]

$$\frac{instructionAt(m, PC) = load\ i}{P \vdash \langle H, \langle m, PC, OS, L\rangle :: FS\rangle \Rightarrow \langle H, \langle m, PC + 1, v :: OS, L\rangle :: FS\rangle}$$

$$Where\ v = L[i]$$

store stores values in the Local Variable Array. It only affects the current method's Frame and requires two things to be executed successfully. First, the Operand Stack has to contain a value at the top. Second, the instruction has to be complemented with a second byte, referred to here as i for index.

The instruction pops the value from the operand and stores it in the Local Variable Array on the index location.

[STORE]

$$\frac{instructionAt(m, PC) = store\ i}{P \vdash \langle H, \langle m, PC, v :: OS, L\rangle :: FS\rangle \Rightarrow \langle H, \langle m, PC + 1, OS, L[i \mapsto v]\rangle :: FS\rangle}$$

return finishes the current method and returns to the method that invoked it. There are two Variants, one for when the method returns a value and another for when it doesn't. It affects both the current method's Frame (top of the Frame Stack) and the invoking method's Frame (below the current Frame on the Frame Stack). The instruction that returns a value requires one to be on top of the Operand Stack of the current method's Frame.

In variant one the value is popped from the Operand Stack of the current method's Frame before that Frame is popped from the Frame Stack. The integer is then pushed to the Operand Stack of the invoking method, the invoking method is now the current method, and its PC is incremented. This instruction also exists for short and reference values. Variant two is the same, but without the value moving between frames.

[RETURN1]

$$\frac{instructionAt(m, PC) = return}{\begin{array}{l}P \vdash \langle H, \langle m, PC, v :: OS, L\rangle :: \langle m', PC', OS', L'\rangle :: FS\rangle \Rightarrow \\ \langle H, \langle m', PC' + 1, v :: OS', L'\rangle :: FS\rangle\end{array}}$$

[RETURN2]

$$\frac{instructionAt(m, PC) = return}{\begin{array}{l}P \vdash \langle H, \langle m, PC, OS, L\rangle :: \langle m', PC', OS', L'\rangle :: FS\rangle \Rightarrow \\ \langle H, \langle m', PC' + 1, OS', L'\rangle :: FS\rangle\end{array}}$$

new creates new objects and affects the current method's Frame and the Heap. It requires the instruction to be complemented by a class reference. It is a reference to the class that is to be instantiated into a new Object.

The instruction uses the information from the Class to create an Object on the Heap, the instance values are set to their default values. An object reference is pushed to the Operand Stack and the PC is incremented.

[NEW]

$$\frac{instructionAt(m, PC) = new\ cr}{P \vdash \langle H, \langle m, PC, OS, L\rangle :: FS\rangle \Rightarrow \langle H', \langle m, PC+1, v_{ObjRef} :: OS, L\rangle :: FS\rangle}$$

$$\text{Where } cr \in ClassRef,\ v_{ObjRef} \in ObjRef,\ v_{ObjRef} \notin domain(H),\ mathitH' = H[v_{ObjRef} \mapsto o],\ o \in Object,\ o.ClassRef = cr$$

getfield gets the field values of an Object. It requires the instruction to be complemented with a *FieldRef* that points to a specific field of a Class. The instruction also requires an object reference on the Operand Stack that points to the part of the Heap containing the Object. The instruction affects the current method's Frame and reads from the Heap.

The instruction pops the object reference and the index from the Operand Stack and instead pushes the requested Field value to the Operand Stack and increments PC.

[GETFIELD]

$$\frac{instructionAt(m, PC) = getfield\ fref,\ \ o = H(v_{ObjRef}),\ value = o.Field(fref)}{P \vdash \langle H, \langle m, PC, v_{ObjRef} :: OS, L\rangle :: FS\rangle \Rightarrow \langle H, \langle m, PC+1, value :: OS, L\rangle :: FS\rangle}$$

$$\text{Where } v_{ObjRef} \in ObjRef,\ fref \in FieldRef,\ v_{ObjRef} \neq Null$$

putfield functions in the same manner as the *getfield* instruction, but with some variation. The Operand Stack must contain a value beneath the object reference. That value is popped from the Operand Stack with the reference, and put into the object's field and the PC is incremented.

[PUTFIELD]

$$\frac{\begin{array}{c} instructionAt(m, PC) = putfield\ fref, \\ o = H(v_{ObjRef}),\ o' = o[Field \mapsto o.Field[fref \mapsto v]], \\ H' = H[v_{objRef} \mapsto o'] \end{array}}{\begin{array}{l} P \vdash \langle H, \langle m, PC, v_{objRef} :: v :: OS, L\rangle :: FS\rangle \Rightarrow \\ \langle H', \langle m, PC+1, OS, L\rangle :: FS\rangle \end{array}}$$

$$\text{Where } v_{ObjRef} \in ObjRef,\ fref \in FieldRef,\ o \in Object,\ v_{ObjRef} \neq Null$$

invokevirtual is an instruction that allows methods to call other methods. It requires the instruction to be complemented by a Method Signature and that the current method has an object reference on top of the Operand Stack. Any arguments passed to the method are beneath the object reference on the stack. It affects the current method's Frame and the Frame Stack.

The Method Signature refers to the specific method being invoked. A new Frame is made to accompany the new method which is pushed onto the Frame

Stack. The new method becomes the current method while the previous is referred to as the invoking method. The object reference and the arguments have been popped from the invoking method Operand Stack and loaded onto the current method Frame's Local Variable Array, the object reference goes in index 0, the first argument in index 1 and subsequent arguments in subsequent indexes. The Object reference refers to the instance on which the method is being invoked, the arguments are the parameters of the method.

$Super(cl) = Class_{\perp}$
$\{where\ cl = (Name, Field, Class_{\perp}, Method, Program) \in Class$

$$MethodLookup(ms, cl) = \begin{cases} m & \text{if} \quad m \in cl.Method, ms = m.MethodSignature \\ undefined & \text{if} \quad cl = java.lang.Object \\ MethodLookup(ms, Super(cl)) & otherwise \end{cases}$$

[INVOKEVIRTUAL]

$$\frac{instructionAt(m, PC) = invokevirtual\ ms,\ m' = MethodLookup(ms, cl),\ n = |ms.Value|}{\begin{array}{l} P \vdash \langle H, \langle m, PC, v_{objRef} :: arg1 :: arg2 :: ... :: argn :: OS, L\rangle :: FS\rangle \Rightarrow \\ \langle H, \langle m', 0, \epsilon, [0 \mapsto v_{ObjRef}, 1 \mapsto arg1, 2 \mapsto arg2, ..., n \mapsto argn]\rangle :: \\ \langle m, PC, OS, L\rangle :: FS\rangle \end{array}}$$

Where $ms \in MethodSignature,\ cl \in Class,\ o = H(v_{objRef}),\ cl = o.Class,$