Todo list

Aalborg University

Master's Thesis

Securing Java

Rewriting, Modelling and Verification of Java Byte Code

Group:

Christoffer Ndũrũ

Kristian Mikkel Thomsen

Supervisors:

René Rydhof Hansen

Mads Christian Olesen

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science

Software 9th semester

Address: Selma Lagerlöfs Vej 300
9220 Aalborg Øst

Phone no.: 99 40 99 40

Fax no.: 99 40 97 98

Homepage: `http://www.cs.aau.dk`

Project title:
Rewriting, Modelling and Verification of Java Byte Code

Subject:
Securing Java

Project period:
1. Feb 2015 - 30. Jun 2016

Group name:
des107f16

Supervisors:
René Rydhof Hansen

Mads Christian Olesen

Group members:
Christoffer Ndũrũ

Kristian Mikkel Thomsen

Copies: 5

Pages: 31

Appendices: 0 & 0 CD

Finished: xx. Jun 2016

Abstract:

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Preface

Preface goes here.

Signatures

Christoffer Ndũrũ

Kristian Mikkel Thomsen

Contents

CHAPTER 1

Introduction

1.1 Fault Scenarios

We consider two general categories of faults that can occur to a Java Card: *persistent* and *transient* faults. The main difference between these is that the persistent faults will affect the program every run, while the transient faults will only be present for a limited amount of time. Faults can occur when hardware is exposed to radiation sources, e.g. infrared light, laser, heat, physical abuse or cosmic radiation from space. Persistent faults in a piece of hardware, such as system memory, can occur in several way, suchas as a directed fault injection, e.g. a laser beam, targeting a persistent part of memory, such as the EEPROM of a Java Card. This could cause a bit flip in a value that is persistent across power-ups, and thus cause a persistent fault since the wrong value will always be used. If one wishes to create a persistent fault, precision is important, both to strike a persistent part of memory, but also to affect the correct value.

Transient faults do not cause any permanent damage to the hardware. They can cause a temporary bit flip, resulting in a corrupted value, changed control flow to cause unintended behaviour, or a crash of the hardware. The altered behavior will disappear and the fault injection will have to be performed again, if the effect is to be reproduced. Nonetheless, both persistent and transient faults can have fatal consequences, if they strike at the right time and the right place, e.g. for an attacker trying to change a programs control flow and thus execute a sensitive piece of code.

The two categories of fault injection are thus sensitive to two variables, *time* and *place*, to different degrees. For example, an attacker who is trying to alter a constant in a program on a chipped access card, is able to work on the card in private surroundings. He can remove the protective layer on the chip and induce a persistent error on the card. Since the card is offline, the timing of the fault injection does not matter because he can only affect static values on the chip. The fault will still be present when the card is powered on at a later time.

On the other hand, an attacker who wants to change transient properties such as program flow dependent on a non-constant value, is very dependent on both timing and precision of his attack. He has to affect the correct place in memory at just the right time in the program's life cycle to alter the program flow. Table T1-1 illustrates the dependencies of persistent and transient fault injections.

#1 Christoffer: do we have to choose one or do we analyse both?

	Persistent	Transient
Timing		X
Precision	X	X

Table T1-1: Table showing dependencies of induced faults

It is important to note that when attacking the chip offline, the attacker only has access to persistently stored values. When attacking the card in online mode, the attacker also has access to run-time related values, such as user input values stored on the operand stack.

It is also interesting to note that an attacker performing a fault injection on the chip while it is offline, has to leave the card in an uncorrupted state, since it will not boot up correctly if e.g. the *setup()* method of a Java Card was hit by the fault injection, thus corrupting it. This is another reasion *precision* is important. When performing a fault injection on a card which is online, the attacker can strike both persistent values and run-time values . The attacker additionally does not have to leave the card in an uncorrupted state as he would have to when injecting a fault on an offline card. The reason for this is that the attacker might only need to flip a particular bit in e.g. an response APDU packet, or an operand stack value, to trick a card terminal into accepting a transaction. After the card has sent the manipulated packet, the attacker does not care whether the card crashes since the purpose of the attack has been served.

#2 Christoffer: bedre ord er??

#3 Christoffer: this should be explained maybe in a glossary (sub)section?

1.2 Java Card

1.3 Fault Injection

CHAPTER 2

Rewriting a Java Program

2.1 Proposed Solution

We propose a solution which can convert a Java class file to a UPPAAL SMC model, rewrite it to insert fault injection attack countermeasures and recommend one of these countermeasures. The point of the conversion to a model, is to be able to provide guarantees about certain properties of the program. This makes sense when the Java class file is rewritten and countermeasures are inserted, since it is then possible to guarantee that the program has not become less secure with respect to certain properties.

The workflow stages of the solution are illustrated in Figure F2-1. The stages are labelled with numbers 1-5. Their purposes are detailed in the following.

Stage 1 - 2 rewrites the original Java class file to include fault injection countermeasures. The output is one or more modified Java class files depending on the number of countermeasures implemented.

Stage 2 - 3 provides two options, either one can use the rewritten code to perform static analysis with an analysis tool, or one can parse the code through the solution's parser and generate a UPPAAL SMC model.

Stage 3 - 4 modifies the model to include a special fault injection template, which can simulate a fault being injected in the Java program. At this point in the workflow, the model of the rewritten code is also modified to include special synchronisations, guards, updates and locations. These make it possible to verify timed properties of the model and enables the fault template to interact with the model.

Stage 3,4 - 5 is where the solution recommends a fitting countermeasure for the code, based properties it has verified about the rewritten version(s) of the code. These models serve as the input for stage 5. The recommended countermeasure incurs the least static size overhead of the rewritten Java class file but provides the most protection for the largest possible duration of the program's execution.

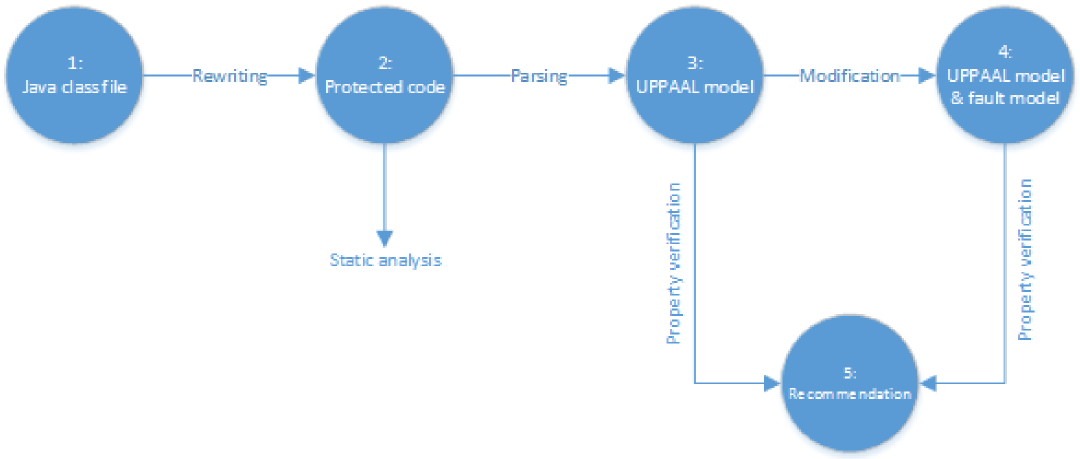


Figure F2-1: The workflow of the solution, from Java class file to UPPAAL SMC model

#4 Christoffer: make better figure

2.2 Program Analysis with Sawja

CHAPTER 3

Semantics

#5 Kristian: hvordan indkludere vi semantiken fra sidste raport?

In this chapter we describe and formalise the language TinyJCL, which contains a variation of the core instructions of the Java Card bytecode language. In this context the term core describes the basic set of instructions from which all other Java Card instructions can be built. We created this language because it is easier to model the fewer instructions in this language, rather than all the instructions in Java Card. The full set of Java Card instructions can be built from combinations of the instructions in TinyJCL. Furthermore, there exists no official formal semantics for the Java Card language. The instructions of TinyJCL are defined as:

$$\begin{aligned} \mathit{Instructions} = \{ & \texttt{NOP}, && \texttt{PUSH}\ v, && \texttt{POP}, \\ & \texttt{ADD}, && \texttt{DUP}, && \texttt{GOTO}\ a, \\ & \texttt{IF\_CMPEQ}\ a, && \texttt{INVOKESTATIC}\ mid, && \texttt{RETURN}, \\ & \texttt{PUTSTATIC}\ fid, && \texttt{GETSTATIC}\ fid, && \texttt{LOAD}\ a, \\ & \texttt{STORE}\ a, && \texttt{INVOKEVIRTUAL}\ mid, && \texttt{PUTFIELD}\ fid, \\ & \texttt{GETFIELD}\ fid, && \texttt{NEW}\ ci && \} \end{aligned}$$

$\mathbb{N}$ is defined as the set of all natural numbers, including zero, and $\mathbb{Z}$ is defined as the set of all integers. In the operational semantics we want to describe values as an integer between a minimum value and a maximum value, $Values = \{x | x \in \mathbb{Z} \wedge x \geq \texttt{INT\_MIN} \wedge x \leq \texttt{INT\_MAX}\}$. In addition we want a notion of addresses which is used to refer to an instruction in a method and mapping to the heap: $\mathit{Addresses} = \mathbb{N}$. A program counter is used to represent the current address $\mathit{ProgramCounters} = \mathit{PC} = \mathit{Addresses}$. Instructions with parameters, such as `PUSH` v, increment the program counter with more than one, since it uses more than one byte.

The program is a sequence of instructions, we denote a program as $P = (i_0, \ldots, i_k)$ where k is the number of instructions in the program. A program consist solely of instructions $P \in \mathit{Programs}$ and $\mathit{Programs} = \{x | x \in \mathit{Instructions}^*\}$. To access instructions we introduce a function accepting a program, method identifier, and a program counter. It returns the instruction in the method of the program at the program counter. The function is defined as:

$$\mathit{inst} = \mathit{Programs} \times \mathit{MethodID} \times \mathit{PC} \rightarrow \mathit{Instructions}$$

$$MethodID = \mathbb{N}$$

To describe a running program we use configurations. A configuration is a 4-tuple consisting of a program, constant pool, heap and a call stack.

$$Conf = Program \times ConstPool \times Heap \times CallStack$$

Executing an instruction means moving from one configuration to another. We will use $\vdash$ to indicate no change in the elements left of $\vdash$. For the semantic rules, no change will occur in program and constant pool e.g.:

$$CP, P \vdash \langle H, CS \rangle \rightarrow \langle H', CS' \rangle$$

Where $CP \in ConstPool$, $H, H' \in Heap$, and $CS, CS' \in CallStack$. We use a shorthand dot notation to access elements of a tuple e.g. $conf.Program$ where $conf \in Conf$, indicates the program used in the configuration $conf$.

The heap can be described as a function which takes a heap address and returns either the address *or* value associated with that address $Heap = Addresses \rightarrow (Addresses + Values)_{\perp}$. $\perp$ represents an undefined value, and is included to describe that *Addresses* can also map to undefined addresses/values in the heap.

The call stack is used to keep track of the current method scope, it is a sequence of stack frames $CallStack = StackFrames^{*}$. A stack frame holds the method id, local variables, operand stack and the program counter for the method.

$$StackFrame = MethodID \times Locals \times OpStack \times PC$$

Local variables are represented by the function $Locals = \mathbb{N} \rightarrow Values_{\perp}$. The operand stack is a sequence of values and addresses $OpStack = (Values + Addresses)^{*}$.

To represent objects we need classes in our language. We represent classes as a 2-tuple with a possible super class and a function for resolving methods: $Class = Class_{\perp} \times Methods$. $Class_{\perp}$ is the super class or $\perp$ in the case of no super class *Methods* is the set of all method identifiers implemented by the class. Object are represented by a 2-tuple with the class and fields of the object:

$$Object = Class \times Fields$$

Fields is a function for resolving the values of class variables:

$$Fields = FieldID \mapsto (Values + Addresses)$$

$$FieldID = \mathbb{N}$$

Finally we make use of a constant pool to resolve static method ids, fields of static classes and class definition when creating new objects:

$$CP = ConstPool = (MethodID \rightarrow \mathbb{N}) + (FieldID \rightarrow Addresses) + (ClassIndex \rightarrow Class)$$

$$ClassIndex = \mathbb{N}$$

3.1 Instruction Semantics

In the following semantics we make use of these abbreviation:

$$\begin{array}{rrr} H, H' \in Heap & CS \in CallStack & ops, ops', ops'' \in OpStack \\ mid, mid', mid'' \in MethodID & loc, loc' \in Locals & pc, pc' \in PC \\ v \in Values & a, objr \in Addresses & fid \in FieldID \\ obj, obj' \in Object & cl \in Class & \end{array}$$

3.1.1 NOP

The instruction has no other effect than incrementing the *pc*.

$$\text{NOP}\ \frac{inst(P, mid, pc) = \texttt{NOP}}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops, pc+1\rangle)\rangle}$$

3.1.2 PUSH

`PUSH` v is used to add the value from the parameter v onto the top of the operand stack. Since the `PUSH` v instruction takes up two bytes due to the parameter, *pc* is incremented by two.

$$inst(P, mid, pc) = \texttt{PUSH}\ v$$

$$\text{PUSH}\ \frac{ops = (x_0, \ldots, x_n) \qquad ops' = (x_0, \ldots, x_n, v)}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops', pc+2\rangle)\rangle}$$

3.1.3 POP

This will remove and discard the top element of the operand stack.

$$inst(P, mid, pc) = \texttt{POP}$$

$$\text{POP}\ \frac{ops = (x_0, \ldots, x_{n-1}, x_n) \qquad ops' = (x_0, \ldots, x_{n-1})}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops', pc+1\rangle)\rangle}$$

3.1.4 ADD

This instruction consumes the two top elements of the operand stack, adding them together and pushes the result back onto the stack.

$$inst(P, mid, pc) = \texttt{ADD} \qquad v = x_{n-1} + x_n$$

$$\text{ADD}\ \frac{ops = (x_0, x_1, \ldots, x_{n-2}, x_{n-1}, x_n) \qquad ops' = (x_0 \ldots x_{n-2}, v)}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops', pc+1\rangle)\rangle}$$

3.1.5 DUP

`DUP` duplicates the top element of the operand stack, leaving two identical elements as the two top elements of the operand stack.

$$inst(P, mid, pc) = \texttt{DUP}$$

$$\text{DUP}\ \frac{ops = (x_0, \ldots, x_n) \qquad ops' = (x_0, \ldots, x_n, x_n)}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops', pc + 1\rangle)\rangle}$$

3.1.6 GOTO

`GOTO` a takes an address as parameter and performs a jump to the specified address.

$$\text{GOTO}\ \frac{inst(P, mid, pc) = \texttt{GOTO}\ a \qquad pc' = a}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops, pc'\rangle)\rangle}$$

3.1.7 IF_CMPEQ

This compares and consumes the two top elements of the operand stack. If they are equal it will make a jump to the address given as a parameter, otherwise it will increment *pc* by two.

$$inst(P, mid, pc) = \texttt{IF\_CMPEQ}\ a$$

$$pc' = \begin{cases} a, & \text{if } x_{n-1} = x_n \\ pc + 2, & \text{otherwise} \end{cases}$$

$$\text{IF\_CMPEQ}\ \frac{ops = (x_0, \ldots, x_{n-2}, x_{n-1}, x_n) \qquad ops' = (x_0, \ldots, x_{n-2})}{CP, P \vdash \langle H, (CS, \langle mid, locals, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, locals, ops', pc'\rangle)\rangle}$$

3.1.8 INVOKESTATIC

`INVOKESTATIC` *mid* is used to call a static method. This involves adding a new stack frame on the call stack. The parameters of the methods are stored in local variables of that stack frame. These parameters are read from the operand stack. The number of parameters, *pn*, are found in the constant pool.

$$inst(P, mid, pc) = \texttt{INVOKESTATIC}\ mid' \qquad CP(mid') = pn$$

$$ops = (x_0, \ldots, x_n) \qquad ops' = (x_0, \ldots, x_{n-pn})$$

$$\text{INVOKESTATIC}\ \frac{loc' = [0 \mapsto x_{n-pn}, \ldots, pn \mapsto x_n])}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow}$$

$$\langle H, (CS, \langle mid, loc, ops', pc\rangle, \langle mid', loc', \epsilon, 0\rangle)\rangle$$

3.1.9 RETURN

`RETURN` is used when returning from a method. The result of a `RETURN` depends on the state of the operand stack when called. If the operand stack is not empty the top element will be the return value.

$$\text{RETURN}\frac{inst(P, mid', pc') = \texttt{RETURN} \qquad ops = (x_0, \ldots, x_n) \\ ops' \neq \epsilon \qquad ops' = (x'_0, \ldots, x'_n) \qquad ops'' = (x_0, \ldots, x_n, x'_n)}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle, \langle mid', loc', ops', pc'\rangle)\rangle \Rightarrow \\ \langle H, (CS, \langle mid, loc, ops'', pc + 1\rangle)\rangle}$$

In following case, where the operand stack is empty, it will return without adding an element to the previous frame's operand stack.

$$\text{RETURN VOID}\frac{inst(P, mid', pc') = \texttt{RETURN} \qquad ops' = \epsilon}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle, \langle mid', loc', ops', pc'\rangle)\rangle \Rightarrow \\ \langle H, (CS, \langle mid, loc, ops, pc + 1\rangle)\rangle}$$

3.1.10 PUTSTATIC

This is used to write a value to a class variable on the heap.

$$\text{PUTSTATIC}\frac{inst(P, mid, pc) = \texttt{PUTSTATIC}\ fid \\ CP(fid) = a \qquad H' = H[a \mapsto v] \\ ops = (x_0, \ldots, x_n, v) \qquad ops' = (x_0, \ldots, x_n)}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H', (CS, \langle mid, loc, ops', pc + 2\rangle)\rangle}$$

3.1.11 GETSTATIC

This reads the value of a class variable on the heap.

$$\text{GETSTATIC}\frac{inst(P, mid, pc) = \texttt{GETSTATIC}\ fid \\ CP(fid) = a \qquad H(a) = v \qquad v \neq \perp \\ ops = (x_0, \ldots, x_n) \qquad ops' = (x_0, \ldots, x_n, v)}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops', pc + 2\rangle)\rangle}$$

3.1.12 LOAD

`LOAD` i is used to load the value of a local variable onto the operand stack.

$$\text{LOAD}\;\frac{\mathit{inst}(P,\mathit{mid},\mathit{pc}) = \texttt{LOAD}\ i \qquad \mathit{loc}(i) = v \qquad v \neq \bot \qquad \mathit{ops} = (x_0 \ldots x_n) \qquad \mathit{ops}' = (x_0 \ldots x_n, v)}{CP, P \vdash \langle H, (CS, \langle \mathit{mid}, \mathit{loc}, \mathit{ops}, \mathit{pc}\rangle)\rangle \Rightarrow \langle H, (CS, \langle \mathit{mid}, \mathit{loc}, \mathit{ops}', \mathit{pc}+2\rangle)\rangle}$$

3.1.13 STORE

This will store a new value in a local variable.

$$\text{STORE}\;\frac{\mathit{inst}(P,\mathit{mid},\mathit{pc}) = \texttt{STORE}\ i \qquad \mathit{loc}' = \mathit{loc}[i \mapsto x_n] \qquad \mathit{ops} = (x_0,\ldots,x_{n-1},x_n) \qquad \mathit{ops}' = (x_0,\ldots,x_{n-1})}{CP, P \vdash \langle H, (CS, \langle \mathit{mid}, \mathit{loc}, \mathit{ops}, \mathit{pc}\rangle)\rangle \Rightarrow \langle H, (CS, \langle \mathit{mid}, \mathit{loc}', \mathit{ops}', \mathit{pc}+2\rangle)\rangle}$$

3.1.14 INVOKEVIRTUAL

`INVOKEVIRTUAL` is similar to `INVOKESTATIC` but in addition an object reference from the operand stack is stored as the first local variable, and the method for the actual class is resolved by a method lookup, inspired by [2]. For this we introduce two functions *signa*, and *methodLookup*. $\mathit{signa} = \mathit{MethodID} \rightarrow \mathit{Signature}$, where *Signature* is the method's signature e.g. name and parameters. And *methodLookup* used to lookup the intended method identifier, either from the class itself or a super class, defined as:

$$\mathit{methodLookup}(\mathit{mid}, \mathit{cl}) = \begin{cases} \bot & \mathit{if}\ \mathit{cl} = \bot \\ \mathit{mid}' & \mathit{if}\ \mathit{mid}' \in \mathit{cl}.\mathit{Methods} \wedge \mathit{signa}(\mathit{mid}') = \mathit{signa}(\mathit{mid}) \\ \mathit{methodLookup}(\mathit{mid}, \mathit{cl}.\mathit{Class}) & \mathit{otherwise} \end{cases}$$

$$\text{INVOKEVIRTUAL}\;\frac{\begin{array}{c}\mathit{inst}(P,\mathit{mid},\mathit{pc}) = \texttt{INVOKEVIRTUAL}\ \mathit{mid}' \qquad CP(\mathit{mid}') = \mathit{pn} \\ \mathit{ops} = (x_0,\ldots,x_n,\mathit{objr},p_1,\ldots,p_{pn}) \qquad \mathit{ops}' = (x_0,\ldots,x_n) \\ \mathit{methodLookup}(H(\mathit{objr}).\mathit{Class},\mathit{mid}') = \mathit{mid}'' \qquad \mathit{mid}'' \neq \bot \\ \mathit{loc}' = [0 \mapsto \mathit{objr}, 1 \mapsto p_1, \ldots, \mathit{pn} \mapsto p_{pn}]\end{array}}{\begin{array}{c}CP, P \vdash \langle H, (CS, \langle \mathit{mid}, \mathit{loc}, \mathit{ops}, \mathit{pc}\rangle)\rangle \Rightarrow \\ \langle H, (CS, \langle \mathit{mid}, \mathit{loc}, \mathit{ops}', \mathit{pc}\rangle, \langle \mathit{mid}'', \mathit{loc}', \epsilon, 0\rangle)\rangle\end{array}}$$

#6 Kristian: private vs public calls

3.1.15 PUTFIELD

`PUTFIELD` fid takes an object reference and a value from the top of the operand stack and stores the value in a specific field in the object.

$$\text{PUTFIELD}\ \frac{\begin{array}{c} inst(P, mid, pc) = \texttt{PUTFIELD}\ fid \qquad H(objr) = obj \\ H' = H[objr \mapsto obj'] \qquad obj' = obj.Fields[fid \mapsto v] \\ ops = (x_0, \ldots, x_n, objr, v) \qquad ops' = (x_0, \ldots, x_n) \end{array}}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H', (CS, \langle mid, loc, ops', pc+2\rangle)\rangle}$$

3.1.16 GETFIELD

`GETFIELD` fid reads and consumes an object reference from the operand stack, and reads the value of the specified field in the object which is then stored on the operand stack.

$$\text{GETFIELD}\ \frac{\begin{array}{c} inst(P, mid, pc) = \texttt{GETFIELD}\ fid \\ obj = H(objr) \qquad v = obj.Fields(fid) \\ ops = (x_0, \ldots, x_n, objr) \qquad ops' = (x_0, \ldots, x_n, v) \end{array}}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops', pc+2\rangle)\rangle}$$

3.1.17 NEW

`NEW` ci creates a new object on the heap as well as pushing an object reference to the operand stack.

$$\text{NEW}\ \frac{\begin{array}{c} inst(P, mid, pc) = \texttt{NEW}\ ci \qquad CP(ci) = cl \\ obj = \langle cl, fields\rangle \qquad fields \in Fields \\ H(objr) = \perp \qquad H' = H[objr \mapsto obj] \\ ops = (x_0, \ldots, x_n) \qquad ops' = (x_o, \ldots, x_n, objr) \end{array}}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle H', (CS, \langle mid, loc, ops', pc+2\rangle)\rangle}$$

3.2 Fault Semantics

We now introduce a fault model formalising the fault injections which can happen. This serves the purpose of showing exactly how a certain fault affects the Java Card, whether it be a program flow change or a change in the memory.

To describe the fault semantics, we represent $Values$ and $Addresses$ as bit strings. We then use a Hamming distance of one, $\equiv_1$, to represent a bit flip, so $a \equiv_1 b$ means a differs from b with only one bit. It can also be stated as

$$\exists x \in \mathbb{N} : a \equiv_1 b \Rightarrow a = b \oplus 2^x$$

where $\oplus$ is the binary `XOR` operation. It can be expressed similarly with sequences

$$A \equiv_1 B \Rightarrow e \neq e' | A = (x_0, \dots, x_{n-1}, e, x_{n+1} \dots, x_m) \wedge B = (x_0, \dots, x_{n-1}, e', x_{n+1} \dots, x_m)$$

where n is the position of the differentiating element in the sequences A and B.

3.2.1 DATAFAULT

A data fault can occur three places: the operand stack, the local variables or the heap. These faults are formalised in the `DF_OPS`, `DF_LOC` and `DF_HEAP` rules respectively.

$$\text{DF\_OPS}\;\frac{ops = (x_0, \dots, x_n, \dots, x_m) \qquad ops' = (x_0, \dots, v, \dots, x_m) \qquad v \equiv_1 x_n \qquad 0 \leq n \leq m}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc \rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops', pc \rangle)\rangle}$$

$$\text{DF\_LOC}\;\frac{fid \in loc \qquad v = loc(fid) \qquad v' \equiv_1 v}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc \rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc[fid \mapsto v'], ops, pc \rangle)\rangle}$$

$$\text{DF\_HEAP}\;\frac{a \in H \qquad v = H(a) \qquad v' \equiv_1 v}{CP, P \vdash \langle H, CS \rangle \Rightarrow \langle H[a \mapsto v'], CS \rangle}$$

3.2.2 PROGRAMFLOWFAULT

This fault causes a change in the program flow of the applet. There are two cases: In the first case, either the fault changes the program counter, which has the consequence of changing which instruction is to be executed next. But since we have defined the program counter to only span locally within the method body, `PFF_PC` only describes a change in program flow within the method body. In the second case, `PFF_M` describes a fault which changes the method identifier, *mid*, of the method to be executed. The fault described by `PFF_M` will change the program flow to another method, outside of the current stack frame, but at the same program counter value within the new method's stack frame.

$$\text{PFF\_PC}\;\frac{pc' \equiv_1 pc}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc \rangle)\rangle \Rightarrow \langle H, (CS, \langle mid, loc, ops, pc' \rangle)\rangle}$$

$$\text{PFF\_M}\;\frac{mid' \equiv_1 mid}{CP, P \vdash \langle H, (CS, \langle mid, loc, ops, pc \rangle)\rangle \Rightarrow \langle H, (CS, \langle mid', loc, ops, pc \rangle)\rangle}$$

3.2.3 INSTRUCTIONFAULT

This fault causes a change in an instruction, e.g. changing a `ADD` to a `POP` by changing a single bit in the instruction.

$$\text{INST\_F}\;\frac{\begin{array}{c} inst(P, mid, pc) = I \qquad P \equiv_1 P' \\ I, I' \in Instructions \qquad I' \neq I \qquad inst(P', mid, pc) = I' \end{array}}{\langle CP, P, H, (CS, \langle mid, loc, ops, pc\rangle)\rangle \Rightarrow \langle CP, P', H, (CS, \langle mid, loc, ops, pc\rangle)\rangle}$$

CHAPTER 4

UPPAAL and Formal Verification

4.1 Property Verification

UPPALL has its own query language used to verify properties of a model[1, p. 7]. The language is a simplified version of timed computation tree logic. UPPAAL's query language consists of *state formulae* and *path formulae*. The path formulae can be categorised into three categories: reachability, safety and liveness.

State formulae A state formula is an expression which can be evaluated for a state, without looking at the mode, e.g. $i \geq 42$. This formula asks whether it is true that i is greater than or equal to 42 in a given state. State formulae also allow one to verify whether a process is in a given location using an expression of the form `P.l`, where `P` is a process and `l` is a location in the process.

A deadlock is described using a special state formula, `deadlock`, and is satisfied for all states which deadlock.

Reachability properties express the notion that a state formula, φ, can *possibly* be satisfied on some path, going from the initial location of the model. In UPPAAL it is expressed as `E<>`φ. This could for example be used to verify whether a variable `i` in the model, along some path going from the initial location will have the value 2 by querying the model with `E<>i == 2`.

These types of properties are often verified as a part of a sanity check of a modelled system[1, p. 8]. For example, when modelling a communication protocol, one can verify that a sender will be able to send a message and that a receiver will be able to receive a message, at some point. Though this does not give any guarantee that a message is always delivered or received, it makes sense to make sure to check whether it eventually *can* be.

Safety properties state that "something bad will never happen". In other words, every state in a model will invariantly satisfy φ. This is useful e.g. if one had created a model of a train track network, and two trains should never be at the same crossing at the same time. Such an invariant safety property is expressed in UPPAAL as `A[]`φ, where the state formula, φ, would express that the two trains are never at the same crossing at the same time.

A variant of this safety property, is one that expresses that "something will possibly never happen", e.g. when playing a game, a safe state would be one where a winning move is always possible. This is expressed in UPPAAL as `E[]`φ, which states that there should exist a maximal path[1], where φ is always true.

Liveness property state that "something will eventually happen". This could for example be that when a coffee maker is turned on, it will eventually output coffee. It is expressed in UPPAAL as `A<>`φ, and means that φ is eventually satisfied.

A variation of this liveness property, is the *leads to* property, written as $\varphi \leadsto \psi$. It is expressed in a UPPAAL query as φ `-->` ψ, and means that if φ is satisfied, ψ will eventually be satisfied, e.g. whenever a message has been sent, it will eventually be received.

4.2 SMC

extra edges for waiting

[1]A maximal path, is a path that is either infinite or the last state has no outgoing edges that can be traversed.

CHAPTER 5

Building a UPPAAL Model

5.1 Program Representation

When translating a program to a UPPAAL of the program. Several representations are possible, depending on what one wants to show. One could for example represent a program merely in terms of program flow if a simulation of a disruption of the program flow is to be shown, e.g an error in the program counter. One could also include the data flow in the program if a simulation of a corruption of a memory value is to be shown. These are just a few examples and many representations can be chosen.

We have chosen the later and model the program in terms of program flow and data flow, so that we can simulation disruptions in the programs execution flow.

5.1.1 Representation Details

The program simulation is based on TinyJCL semantics, most Java bytecodes can be translated directly to TinyJCL at the loss of type information.

When representing Java bytecode in UPPAAL we have chosen to represent an instruction, such as `aload a` and `dup`, as UPPAAL locations. This implies that a change in the program counter is a change of the location. In turn this means that when an instruction is to be executed the change to the program configuration *Conf* from Chapter 3 occurs on the edge to next location.

```
public class Sample{
    public static void main(String[] args) {
        for (String a : args)
        {
            System.out.print(a);
        }
    }
}
```

Listing 5.1: Jave code sample.

```
public static void main(java.lang.String[]);
 Code:
   0: aload_0
   1: astore_1
   2: aload_1
   3: arraylength
   4: istore_2
   5: iconst_0
   6: istore_3
   7: iload_3
   8: iload_2
   9: if_icmpge     31
  12: aload_1
  13: iload_3
  14: aaload
  15: astore        4
  17: getstatic     #2                  // Field java/lang/System
     .out:Ljava/io/PrintStream;
  20: aload         4
  22: invokevirtual #3                  // Method java/io/
     PrintStream.print:(Ljava/lang/String;)V
  25: iinc          3, 1
  28: goto          7
  31: return
```

Listing 5.2: Bytecode sample.

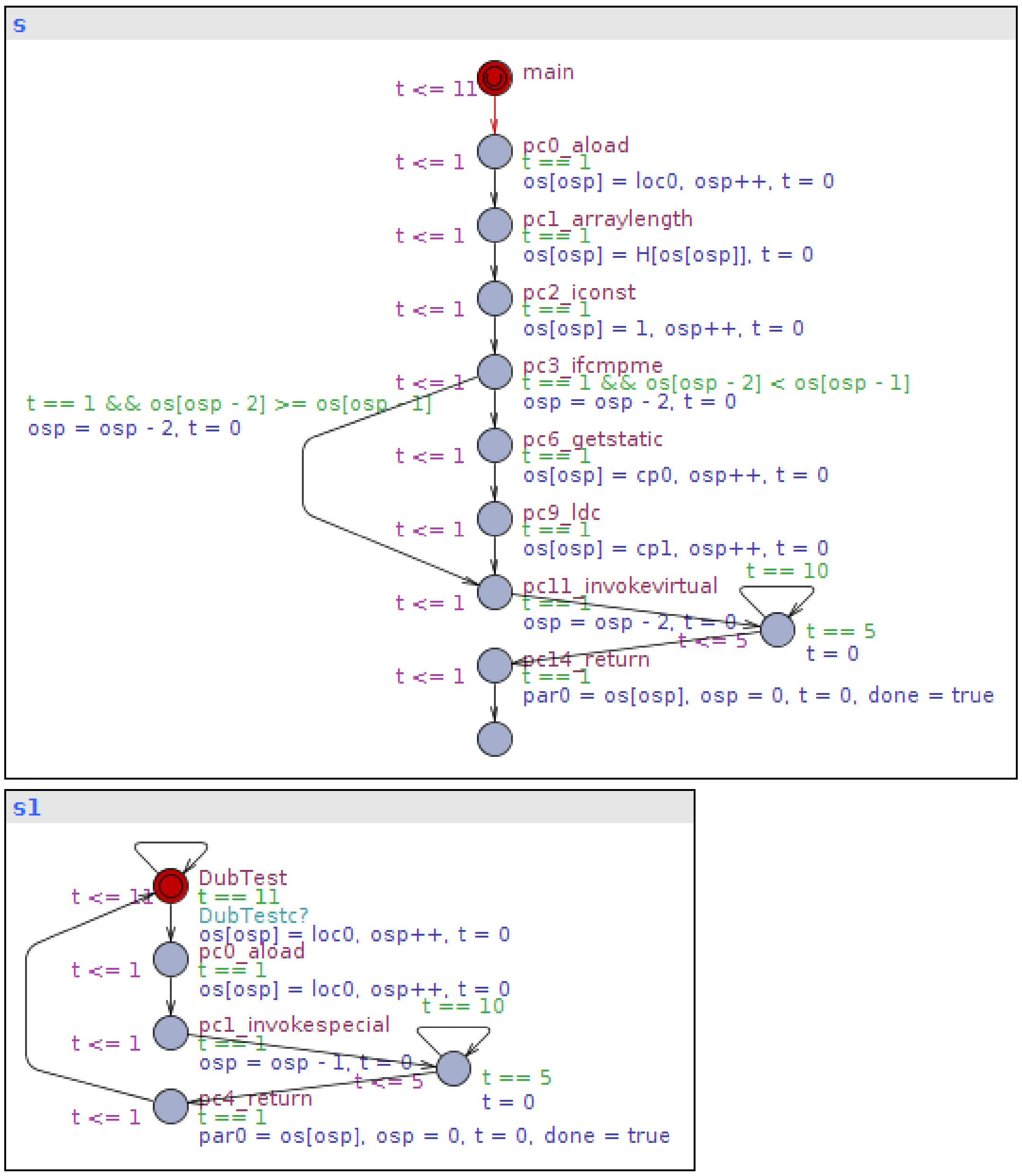


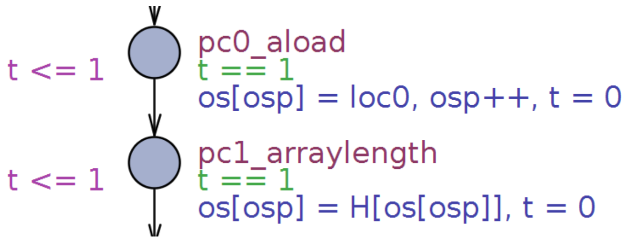
Figure F5-1: Auto generated model, work in progress.

#7 Kristian: update the model for the new sample

Simple Instructions

```
 0. aload 0
 1. arraylength
 ...
```

(a) Java Bytecode Sample.



(b) Generated model from Sample.

Figure F5-2: Java bytecode and corresponding UPPAAL model.

#8 Kristian: lav ny model til nyt sample

Figure F5-2 show how two Java Bytecode instructions are represented in UPPAAL. On the Left we see the Java bytecode, the first line with program counter 0 we have the `aload 0` instruction. `aload 0` pushes a reference to the top of the operation stack from local variables at position zero, then increments the operation stack pointer and program counter.

In UPPAAL the location `pc0_aload` represent `aload 0`. The UPPAAL model is seen in Figure F5-2b. We simulate execution time with the location invariant `t <= 1` and guard `t == 1` on the edge leading to the next location. The guard is found right below the location name right of the edge and invariant is to the left of the edge. In this sample we defined the execution time as 1 time-unit.

In the update on the edge seen below the guard, we simulate the data flow by assigning the value of the local variable `loc0` to the top of the operand stack `os` represented by operand stack pointer `osp`. `osp` is incremented as the operand stack grows, the increment of the program counter is simulated by the edge itself.

Jumps and Branches

For the majority of instructions the program counter is set to the next instruction after execution, but for a jump with `goto a` the edge goes to the instruction with the program counter corresponding with value of `a`.

Conditionals such as `if_cmpeq a` is the only instruction that is modelled by a location having two outgoing edges , one to the next instruction and one for the program counter of a. On these edges the guard is used to determine which of the edges is to be traversed.

#9 Kristian: insert example

Method Calls

Method calls are represented by three additions to the model. These additions consist of locations, but they do not have any associated program counter since they are not a part of the original program.

The first is an addition of a location in the template of the `main` method. This location captures the notion that this is the start of the program.

The second is a new location in the caller for every method call it performs. This makes it possible to simulate parameter passing, as well as control transfer when waiting for a callee to return control to the caller after a method call. The simulation of the caller remains in this location until the callee returns control, after its simulation has finished. This control transfer is modeled with a synchronisation on the edge going from the new state in the caller and back to its original control flow.

#10 Christoffer: insert ref to figure

The third is an addition of two additional states in every template, except for the `main` template. The first, initial, location serves dual purposes: it enables the control transfer from the caller to itself by synchronisation, and simulates passing of arguments into the method from the caller. The second location is the *Done* location, where the simulation ends up when it has finished its simulation. This is where control is transferred back to the caller.

#11 Christoffer: show how parameters are passed in figure

#12 Christoffer: rewrite this

5.2 Modelling a Fault Injection

To simulate a single bit flip occurring in the program's execution a special fault template is introduced. The template calculates a random value between 0 and the maximum possible global clock value, which represents when in the programs execution a fault happens. The random value is assigned to a global variable in the UPPAAL system.

Every instruction in the Java bytecode is represented by a location, and has an associated program counter. There are edges from each location going to the locations which can be reached if one bit is flipped in the program counter. These edges have guards which check whether the time the fault is injected, corresponds to the global clock at the time the model simulation is at that particular edge. If it is, the guard will allow the edge to be traversed. There are no fault edges going back to the added locations described in Section 5.1.1, since these are not a part of the original program and therefore do not have an associated program counter.

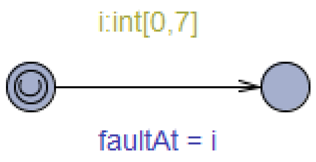


Figure F5-3: The UPPAAL template which performs a bit flip in the program counter

#13 Christoffer: update figure to use fault at between 0 and global clock

5.2.1 Opstack complications

Under normal execution the value of the operand stack pointer can be known at compile time

#14 Kristian: double check, etv find source

. This leaves us with two ways to simulate it, use a operand stack pointer to point at the top element or staticly define the opstack element to be accessed for each instruction.

Under normal execution there is to difference between the two approaches, but then introducing fault models as PC_Fault and INST_Fault it is possible to change the behaviour enough to let a operand stack pointer point to out of the operand stack.

CHAPTER 6

Conclusion

6.1 Future Work

APPENDIX A

Sample Model

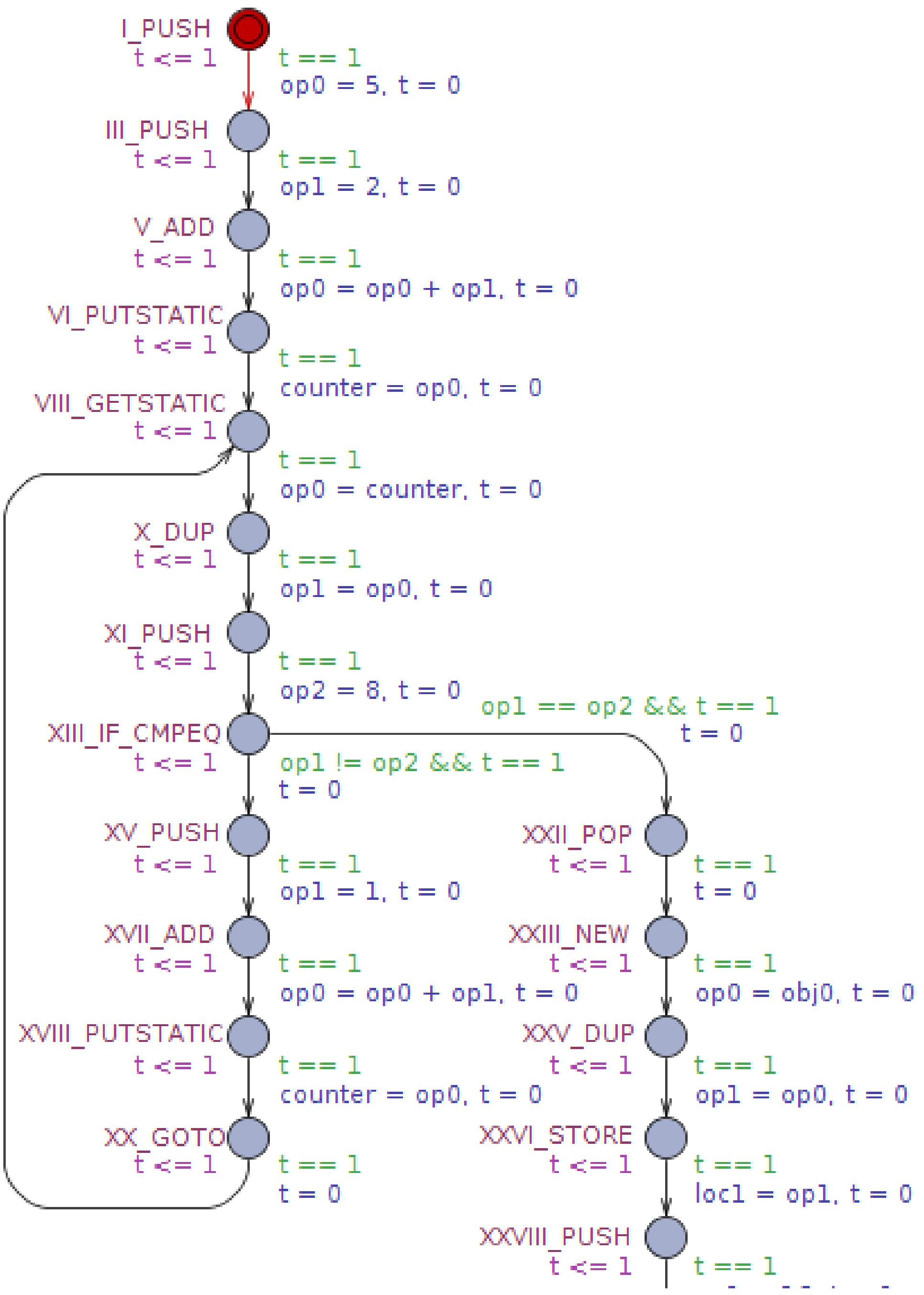


Figure A.1: Model of full TinyJCL

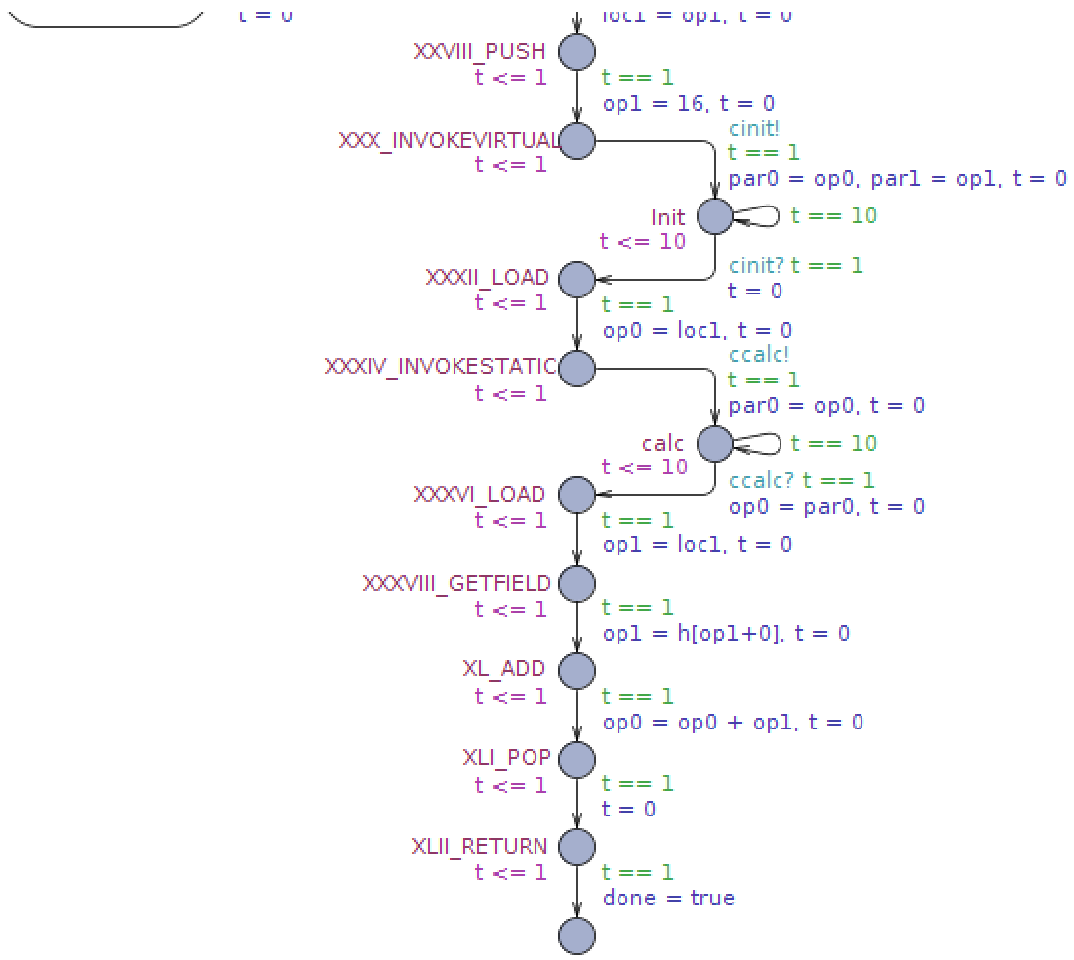


Figure A.2: Model of full TinyJCL

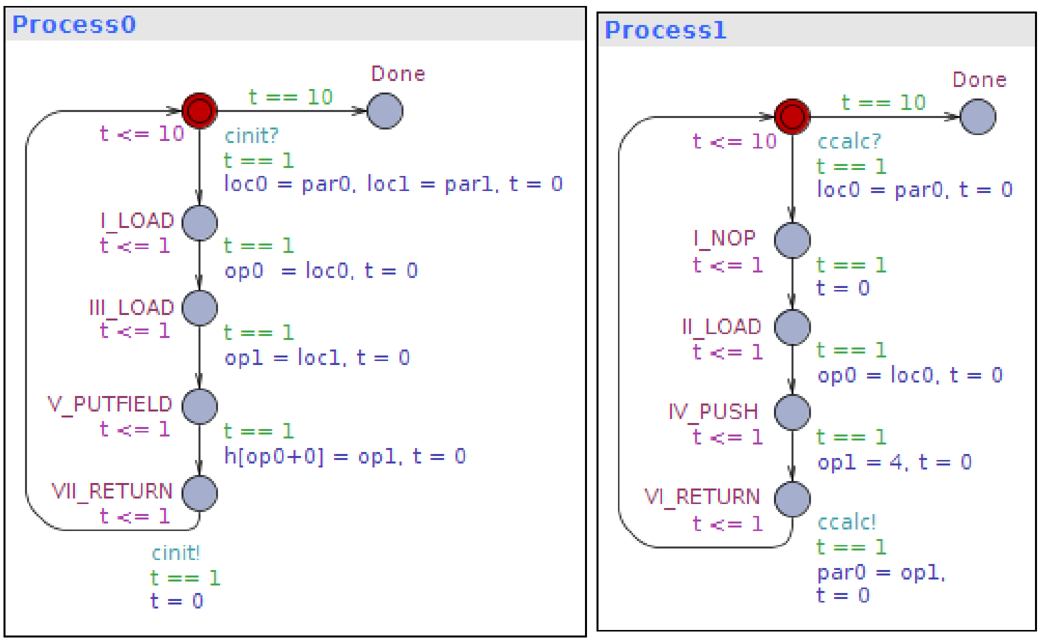


Figure A.3: Model of full TinyJCL

Bibliography

[1] Alexandre David Gerd Behrmann and Kim G. Larsen. *A Tutorial on Uppaal 4.0*. `http://www.uppaal.com/admin/anvandarfiler/filer/uppaal-tutorial.pdf`. Visited 12. March 2015.

[2] Henrik Søndberg Karlsen, Erik Ramsgaard Wognsen, Mads Chr Olesen, and René Rydhof Hansen. Study, formalisation, and analysis of dalvik bytecode. *Informal proceedings of The Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2012)*, 2012.